Advanced Programming Techniques

FOR THE

APPLE IIGS TOOLBOX

Morgan Davis and Dan Gookin

An extensive collection of proven strategies for putting the power of the Apple IIGS to work for all C, Pascal, and machine language programmers.

COMPUTE!
LIBRARY
SELECTION

Advanced Programming Techniques for the APPLE IIGS TOOLBOX

Morgan Davis and Dan Gookin

COMPUTE! Publications, Inc. abc
A Capital Cities/ABC, Inc. Company
Greensboro, North Carolina

Cover design: Lee Noel, Jr.
Editor: Robert Bixby

Copyright 1988, COMPUTE! Publications, Inc. All rights reserved.

Reproduction or translation of any part of this work beyond that permitted by Sections 107 and 108 of the United States Copyright Act without the permission of the copyright owner is unlawful.

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-87455-130-7

The author and publisher have made every effort in the preparation of this book to ensure the accuracy of the information and programs. However, the information and programs in this book are sold without warranty, either express or implied. Neither the authors nor COMPUTE! Publications, Inc. will be liable for any damages caused or alleged to be caused directly, indirectly, incidentally, or consequentially by the information or programs in this book.

The opinions expressed in this book are solely those of the authors and are not necessarily those of COMPUTE! Publications, Inc.

COMPUTE! Publications, Inc., Post Office Box 5406, Greensboro, NC 27403, (919) 275-9809, is a Capital Cities/ABC, Inc. Company, and is not associated with any manufacturer of personal computers. Apple is a registered trademark and Apple IIGS is a trademark of Apple Computer, Inc.

APW C is available only to members through Apple Programmer's and Developer's Association, 290 SW 43rd St., Renton, WA 98055.

TML Pascal is a product of TML Systems, Inc., 4241 Baymeadows Rd., Suite 23, Jacksonville, FL 32217.

Contents

Foreword . . . . *v*

Chapters

1. Introduction 1

2. Programming Subtleties 9

3. How Programs Work 25

4. About the Toolbox 43

5. A Matter of Language 61

6. The DeskTop 77

7. Memory Management 95

8. Pull-Down Menus 113

9. Windows 145

10. Dialog Boxes 187

11. Controls 241

12. Interrupts 271

13. Desk Accessories 309

14. ProDOS 329

Appendices

A. Apple's Human Interface Guidelines 371

B. Tool Sets in the Apple IIGS Toolbox 384

C. Error Handling 391

D. Error Codes 396

E. Event and TaskMaster Codes 401

F. QuickDraw II Color Information 408

Index 411

Foreword

If you have a solid understanding of machine language, Pascal, or C, you'll find *Advanced Programming Techniques for the Apple IIGS Toolbox* invaluable in helping you to improve your Apple IIGS programming skills. This book examines in detail the structures and procedures necessary to make the Apple IIGS perform for you. Although the machine has been available for over a year and a half, the programming market for the Apple IIGS is still wide open. Programs that take full advantage of the machine's capabilities have only begun to appear.

"This book is not for the beginner," the authors warn early in the first chapter. But intermediate- to advanced-level programmers will find *Advanced Programming Techniques for the Apple IIGS Toolbox* packed with solid information on this fast-selling machine. This book delves into the intricacies of the powerful set of libraries known collectively as the Toolbox.

The program examples given here are ready to be merged with your own program code, giving your programs greater flexibility within the IIGS operating system. Mirroring the flexibility of the machine, this book provides a nonlinear approach that allows you to turn to your area of immediate interest, begin learning the things you need to know, and produce the program you're trying to write, in your choice of languages. Along the way, you'll learn about the other languages and the inner workings of the operating system.

Inside this book is information covering DeskTop applications, the mouse, pull-down menus, windows, dialog boxes, controls, special applications like interrupts, and the use of ProDOS 16. You'll also find a condensed version of Apple's Human Interface Guidelines as well appendices packed with technical information.

Advanced Programming Techniques for the Apple IIGS Toolbox is a treasure trove of inside information of all kinds. You'll value its insights into this exciting machine. This is the book every intermediate- to advanced-level programmer needs to enhance his or her programming skills.

Chapter 1

Introduction

To use this book, you should have an assembler, or you should have a Pascal or C language compiler. The software mentioned in this book are the *APW (Apple Programmer's Workshop)* C and machine language development kit and *TML Pascal*. These programs and their associated utilities

are available at the addresses given on the copyright page.

This book is intended to be a complement to *COMPUTE!'s Mastering the Apple IIGS Toolbox,* a tutorial on programming the Apple IIGS Toolbox. This book goes more deeply into the intricacies of using this powerful set of libraries to put a professional polish on applications. It's both a reference and a book of advice on designing and building solid programs in machine language, C, and Pascal.

It's assumed that you've read *COMPUTE!'s Mastering the Apple IIGS Toolbox,* or you're already a highly skilled programmer of the Apple IIGS. If so, you're ready to begin a challenging and enjoyable programming adventure. Keep in mind that this book is not for the beginner.

If you haven't read *COMPUTE!'s Mastering the Apple IIGS Toolbox,* or one of the other worthy introductory texts on this computer, you'd be wise to purchase one and read it before venturing further into this book.

This book also assumes you have an Apple IIGS handy to test the routines. Your computer should have at least 512K of RAM, with one or two disk drives. A color monitor is more interesting to look at, but it is not a necessity.

What This Book Is About

This book provides programming advice for the Apple IIGS in three different languages: machine language, Pascal, and C. A solid understanding of one or more of these programming languages is required to be able to grasp the concepts in this book. You can't program the IIGS without them.

Although the Apple IIGS has the same, decade-old, proven BASIC as its ancestors, Applesoft BASIC is not an appropriate language for writing application programs. In fact, the only way to access the advanced techniques of the Apple IIGS from BASIC is by using in-line machine language, a technique that is not recommended, even for the most venturesome programmers.

If you are a BASIC programmer, you might be interested to know that two new BASICs were announced for the Apple IIGS as this book was being prepared. One, from TML Systems of Jacksonville, Florida, and a second from Apple. (There may be more BASICs forthcoming from other developers.) These BASICs are in their "beta test" stage at this writing (which means they are not yet ready for general release; they have too many bugs).

The Toolbox. The key to programming the Apple IIGS is its Toolbox. The Toolbox contains hundreds of routines and functions that provide the core of programming. Programming the Toolbox is a central part of this book. And examples for programming the Toolbox are provided in C, Pascal, and machine language.

This book is not intended to be a Toolbox tutorial. Instead, it was written to acquaint readers, programmers, and Apple enthusiasts with the finer aspects of programming the machine.

The scope of this book is limited to these general areas: the DeskTop, graphics, low-level tools, and other rarely discussed aspects of the Apple IIGS. These areas are well covered and offer an in-depth look at the inner workings of the computer.

Who This Book Is For

This book will benefit Apple IIGS owners who understand machine language, Pascal, or C. Any one of these languages will do, and, after reading a few chapters, you'll probably learn more about the others.

As mentioned above, you'll need an Apple IIGS with at least 512K. The IIGS is currently being sold with 256K. Even Apple admits this isn't enough. Producing a 256K machine was a decision made to keep the base unit as inexpensive as possible. Another decision based on economy was the choice of the 65816 microprocessor, which is only rated at 2.8 MHz. The engineers could have used a faster chip, but it would have added $100 to the price of the computer.

It's recommended that when you buy a memory card for your IIGS, you pack it full of memory. Memory is relatively inexpensive. For the cost of 16K of RAM in 1980 you can easily put over 1024K (one megabyte) of RAM into your IIGS.

Finally, this book is for anyone excited about the Apple IIGS. It's been over a year and a half since the machine was introduced. Exciting and interesting programs are only starting to appear. With the knowledge you'll gain from this book, you'll soon add your own programs to the growing list of applications for the Apple IIGS.

Unlike *COMPUTE!'s Mastering the Apple IIGS Toolbox*, this book doesn't rely upon complete programs to convey ideas. Instead, only small program examples and snippets of code are used. It's assumed you'll be able to put the example pieces together in your own way when creating applications. The examples listed in this book all work and will function in any program you write.

Though you may be tempted to dive into programming without preparation, you'll gain more if you read the text dealing with each program example before cutting and pasting code. While it looks easy and simple, the Toolbox routines have interdependent relationships with each other. To understand how one tool relies upon another, read the text before and after an example. Then you should be able to adapt it successfully to your use.

This book is half reference and half tutorial. The "refertorial" approach makes this book modular. You can start reading at any point. For example, if you'd like to know how to put a custom icon in one of your dialog boxes, turn to the chapter on dialog boxes, and you'll find an example. Replace the graphic in the example with your own, and you'll have a custom icon. This entire book works that way.

On a larger scale, this book is divided into four major parts, each part concentrating on a specific area of programming the Apple IIGS and the Toolbox. Each part is further broken down into individual areas that cover specific topics. Within each area are individual examples and routines you can use to help you understand and program the Apple IIGS.

You can read any section that interests you, provided that you have taken the time to understand the fundamentals. If any information overlaps or is covered elsewhere, you'll be directed to the proper part and section. Most of the groundwork is covered in this, the first section, so naturally, you should take the time to read through the introductory material. When you're finished, you may proceed through the book at your own pace and in any order.

The book is divided as follows:

- The early chapters offer a general introduction. When you're finished reading them, you'll understand how information is presented in this book. For example, one chapter demonstrates how Toolbox routines are documented in this book for all three programming languages. You may skim that section and return to read it in detail if a concept in a later chapter confuses you. You'll also find a great deal of interesting trivia and general background information in this section.

- The middle portion of the book covers DeskTop applications and using the mouse. It details DeskTop programs, pull-down menus, windows, dialog boxes, and controls. This section covers many concepts unique to the IIGS. Don't be surprised if you find yourself referring to this part of the book often.
- The final chapters go into detail on special applications—those features of the Apple IIGS that don't have a category of their own. This part covers such advanced topics such as interrupts and desk accessories. At the end of this section is a chapter on ProDOS 16. Though unrelated to the Toolbox, ProDOS is as much a part of the Apple IIGS as anything mentioned so far. Loading and saving files from and to disk and other file-management techniques are mentioned in the ProDOS chapter.
- The Appendices provide a reference to the first part of the book. You'll find a version of Apple's Human Interface Guidelines in Appendix A. While not an exact duplicate, this version highlights the most important parts of the Human Interface Guidelines, ensuring that your programs will fall in line with Apple's recommendations for all DeskTop applications.

Interesting trivia surrounding the Apple IIGS is just now rising to the surface. Where appropriate, comments and insights have been included in the main body of the text, but when they are tangential or circumstantial to the topic at hand, they will be set aside in boxes. They were included to give a better understanding of Apple IIGS hardware and software construction.

Conventions Used in This Book

Every effort has been made to maintain notations and conventions used in *COMPUTE!'s Mastering the Apple IIGS Toolbox*. For example, the majority of the numbers you'll see in this book are in hexadecimal (base-16) notation. All hexadecimal numbers are preceded by a $ (dollar sign), and they contain the numbers 1–9 and the capital letters *A–F*, which stand for the values 10–15.

There are three types of hexadecimal numbers used in this book: bytes, words, and long words.

A byte value is a two-digit hexadecimal number ranging from $00 through $FF (0–255 decimal). A word value is a four-digit hexadecimal number ranging from $0000 through $FFFF (0–65535 decimal). Words are composed of two bytes, the most significant byte (MSB) and the least significant byte (LSB). In the word value $FACE, $FA is the MSB and $CE is the LSB.

Long words are new to the Apple II. A long word is an eight-digit hexadecimal number equivalent to two words or four bytes. It ranges in value from $00000000 through $FFFFFFFF (0 through 4,294,967,295 decimal). Long words are composed of two words—the high-order word and the low-order word. In the long-word value $00E100A8, $00E1 is the high-order word, and $00A8 is the low-order word. Long words are primarily used in the Apple IIGS to denote memory locations. Refer to the section on memory addressing in the next chapter for details.

Though not a type of number (or size), the Toolbox uses *logical*, or *Boolean*, values to represent the true or false result of certain operations. A true value is any value not equal to 0. Commonly, true is set to the hexadecimal word value of $8000. A false value is 0.

Logical True = $8000 or any nonzero value
Logical False = $0000

When the Toolbox returns a logical true or false value, the actual numbers returned are as listed above. As might be expected, there are times when the computer breaks this rule and returns 0 for true and a nonzero value for false. When this happens, a note will be provided to warn you about it.

One final convention concerns the program listings in this book. Line numbers are included with all program listings above a certain size. Unless specified, the line numbers are not to be entered (when you type in the examples) or considered as part of the source. The line numbers are intended for use as references from the text. Again, where there are exceptions, they will be noted.

Books Worthy of Note

At this writing, there are few books on the subject of programming the Apple IIGS. However, the books listed below are recommended for anyone interested in programming the Apple IIGS:

- *COMPUTE!'s Mastering the Apple IIGS Toolbox,* Dan Gookin and Morgan Davis (1987, COMPUTE! Publications, ISBN 0-87455-120-X).
- *COMPUTE!'s Apple IIGS Machine Language for Beginners,* Roger Wagner (1987, COMPUTE! Publications, ISBN 0-87455-097-1). Roger wrote the definitive machine language book years ago. This book carries on the tradition.
- *COMPUTE!'s Guide to Sound and Graphics on the Apple IIGS,* William B. Sanders (1987, COMPUTE! Publications, ISBN 0-87455-096-3). Though lacking extensive Toolbox programming examples, this book contains a wealth of information on fundamental Apple IIGS sound and graphics.
- *Apple IIGS Technical Reference,* Michael Fischer (1986, 1987; McGraw-Hill; ISBN 0-07-881009-4). One of the first books to appear on the market, this book is an excellent hardware and software reference to the Apple IIGS. Some of the material is outdated, but it's still worthy.
- *Programming the 65816,* David Eyes and Ron Lichty (1986, Prentice-Hall, ISBN 0-89303-789-3). The ultimate reference to the 65816, with programming examples and the best command reference of any microprocessor book.

Chapter 2

Programming Subtleties

The purpose of this chapter is to acquaint you with some things you should know before attempting to program the Apple IIGS. This information—background material, plus some interesting tidbits—was gathered over a long period of time during visits to the offices of

Apple Computers and through research in virtually every book available on this machine. The material listed here is the distillation of this research. (For more detailed explanations, refer to *COMPUTE!'s Mastering the Apple IIGS Toolbox.*)

How the Apple IIGS Is Different from Other Apples

The Apple II is an "ancient" and honored computer, with a respectable lineage dating back just a little over ten years. Generally speaking, the Apple IIGS is simply the latest incarnation of the Apple II. It has a faster and more powerful microprocessor, better graphics, and advanced sound capabilities, but it can run Apple II software and accommodate Apple II hardware. It also has a *tool set* of programming routines that allow it to mimic its distant cousin, the Macintosh.

In fact, the Apple IIGS is actually one step closer to the Macintosh computer than simply an improvement on the older Apple II design. While the computer is still compatible, the DeskTop extensions, the graphics, and the sound found in the Toolbox routines separate the Apple IIGS from the rest of the Apple II family.

The Apple IIGS is an evolutionary computer in terms of design and implementation. It's difficult to document. The machine's operation is different now from its operation a few months ago. This implies that a shortcut or trick you learn today might not work tomorrow.

Apple is constantly working on the IIGS. Internal modifications are being made, and the firmware and tool sets are constantly being upgraded and modified. Because of this, a warning is offered: Do not stray from the standard.

The Macintosh is another evolutionary machine. The first Macintosh, introduced in 1984, could not compare to the powerful machines Apple makes today. While the Apple IIGS probably won't have the same expensive upgrades the Mac had, it will share the technological advances of its distant relative. Apple has assured its developers that as long as they stick to the standards, their programs will run on all future releases of Apple II computers.

A good example of programmers not sticking to the standards is in the area of the super-hi-res graphics display. Apple has repeatedly warned against finding the screen's secret location in memory. Why? Because it may change in the future. The proper way to use the graphics screen is through the Toolbox. Yet, some developers consider the Toolbox routines slow. For this reason, they prefer to access the screen directly so their programs will work faster. By doing so, they run the risk that in the future they may not work at all.

As long as you adhere to the techniques and programming examples used in this book, you can be assured that your applications will have a long and healthy life—as long as the Apple II series stands. According to Apple, it will last forever.

Here is an abridged history of the Apple computer: The first Apple computer, the Apple I, was actually a circuit-board kit that sold for $666.66 in July 1976.

The Apple II, which came in a case with a keyboard and power supply, was unveiled at the West Coast Computer Faire in April 1977. It came with its own BASIC, 4K of memory, color graphics, and game paddles. The Apple II was available for sale to the general public in June of 1977 for $1,298.

In June 1979, the Apple II+ was introduced. It had an improved ROM, could handle up to 48K of RAM, and retailed for $1,195. In October of that year, the software program *VisiCalc* became available.

The Apple IIe was presented in January 1983. It came with 64K, which could be upgraded to 128K. Also included was a lowercase keyboard option, as well as an 80-column screen. The IIe retailed for $1,395.

The Apple IIc portable was introduced in April 1984. A marketing genius came up with the slogan "Apple II Forever."

In September 1986, the Apple IIGS was introduced. Nine years after the first Apple, the IIGS was priced at $999, came standard with 256K of memory, a keyboard, a mouse, and a mountain of potential.

Graphics

The Apple IIGS contains all the graphics modes of its predecessors, plus a new high-resolution graphics mode. The *super-hi-res* screen is used for all the IIGS graphics and provides a Macintosh-like environment. The responsibility for producing these graphics is given to the Video Graphics Controller (VGC) chip.

The VGC has a big job. Not only does it control the super-hi-res graphics screen, it handles the older Apple II graphics modes, as well as dealing with two different types of interrupts. The VGC allows the Apple IIGS with a color monitor to have a different text, background, and border colors. It also provides foreign language character sets and international video output (for European countries). It's a remarkable piece of engineering.

The following chart shows the Apple IIGS text and graphics screens and their resolutions. The Apple IIe and IIc are both represented by the IIe. The resolution is shown as horizontal pixels by vertical pixels.

Graphics Mode	**Resolution**	**Colors**	**II+**	**IIe**	**Apple IIGS**
Text screen	40 × 24	2 (16 for IIGS only)	*	*	*
Text screen	80 × 24	2 (16 for IIGS only)		*	*
Lo res	40 × 48	16	*	*	*
Double lo res	80 × 48	16		*	*
Hi res	280 × 192	6	*	*	*
Double hi res	560 × 192	1		*	*
Super hi res	320 × 200	16			*
Super hi res	640 × 200	4			*

The 80-column text screen was available to Apple II+ owners via a special 80-column card. However, with the introduction of the Apple IIe, and later the IIc, the 80-column text format became standard.

The lo-res mode displayed graphic "bricks" called pixels (though a pixel usually refers to a small dot). In the hi-res mode, the colors of the pixels and other graphics variations depended on a number of things, most of which are too specific to go into in this book. (A good book on the subject is *COMPUTE!'s Guide to Sound and Graphics on the Apple IIGS* by William B. Sanders.)

Super Hi Res

This book is concerned with the super-hi-res screen on the Apple IIGS. It has two modes: low and high resolution. The high-resolution mode has a pixel resolution of 640 × 200. This mode provides four colors. However, by using a process known as *dithering,* more colors can be produced on the screen. Also, by altering certain attributes of the screen, up to 256 different colors can be produced on one super-hi-res screen.

Questions almost every computer owner asks are "Where is the screen in memory? Is it bitmapped?"

As explained above, this knowledge will not come in handy. However, to be accommodating, a few secrets can be revealed.

At this writing (it will almost certainly change), the super-hi-res graphics screen is located in memory bank $E1, at offset $2000. (Refer to the section on memory management later in this chapter for further explanation of this memory reference.) To activate the screen from Applesoft BASIC, you can type the following (the bracket is the Applesoft prompt):

```
]CALL -151
```

That will put you in the monitor. Type the following to reference memory bank $E1 (the asterisk is the monitor's prompt):

```
*E1/0000
```

Now, activate the super-hi-res mode by putting the byte value $C1 into memory location $C029, the New-video register:

```
*C029:C1
```

That will activate the super-hi-res screen, which implies that from here on you'll be typing "in the dark." Text will be invisible. Sometimes a pretty pattern will appear on the screen. Other times, the data previously on the super-high-res screen can be seen.

Now, any value poked into memory locations $2000–$9CFF will appear on the screen as a pixel, series of pixels, pattern, or color. For example, putting the value $00 into memory location $60B0 might put a black dash near the middle of the screen:

```
*60B0:00
```

You can experiment with your own values (within the proper range of $2000–$9CFF). When you want to return to normal, you must poke a value of $01 back into memory location $C029:

```
*C029:1
```

Or you can type Control-T followed by the RETURN key.

Have fun, but remember the warnings.

The low-resolution mode of the super-hi-res screen has the same vertical resolution (200 pixels) but only half the horizontal resolution of the high-resolution mode. It does, however, have more colors—up to 16—chosen from over 4096 possibilities. By using dithering you can squeeze even more colors out of the low-resolution graphics mode.

You might hear the 640 mode of the super-hi-res screen referred to as "80 columns," and the 320 mode as "40 columns." While this is entirely inaccurate, it does express the appearance of the two modes. In fact, by displaying text on the graphics screen using different fonts, your actual text-screen size varies from 16 rows by 63 columns to 32 rows by 132 columns. (Text is displayed on this screen using a combination of the QuickDraw II and Font Manager tool sets. The size of the text is determined by the font chosen.)

The QuickDraw II tool set in the Apple IIGS Toolbox is responsible for all graphics appearing on the screen. Drawing lines, circles, boxes, arcs, and patterns is easy once you learn how to use the over-250 routines provided by QuickDraw II. By using QuickDraw, you save development time. It eliminates the necessity of writing graphics primitives. The basic code has been written for you. Also, sticking to the QuickDraw routines ensures that your graphics programs will work on and be compatible with all future releases of the Apple IIGS.

Sound

To make the Apple IIGS more competitive and attractive to the marketplace, something had to be done about sound. Sound was one thing the Apple II series of computers barely provided.

For years, Apple II programmers created sound by *bit twiddling*. The speaker has a memory location—$C030. By peeking this location from Applesoft BASIC or by examining this location using assembly language, the speaker could be made to tick (see following box). A rapid succession of ticks produced a tone. By varying the number of ticks and their duration, a chorus of tones could be created. This complicated-yet-simplistic method of producing sound got the job done, yet there had to be a better way.

To tick the speaker in Applesoft BASIC, the PEEK statement is used. PEEK returns the byte value of a specific memory location, in this case $C030, which is 49200 decimal:

```
A = PEEK (49200)
```

The actual value of A can be discarded. By repeatedly reading memory location 49200, as well as varying the interval between PEEKs, the speaker can produce a variety of tones. Note that PEEK's counterpart, POKE, has no audible effect on memory location 49200.

The following program shows how the PEEK statement in Applesoft BASIC can be used to tick the speaker:

```
10 FOR X = 1 TO 10
20 A = PEEK (49200)
30 FOR T = 1 TO 10 : NEXT T
40 A = PEEK (49200)
50 NEXT X
```

The two PEEK statements in lines 20 and 40 tick the speaker. Line 30 contains a delay that produces the pitch of the tone: Increase the delay, and the pitch deepens; decrease the delay, and a higher pitch is produced. The main FOR-NEXT loop between lines 10 and 50 sets the duration of the tone.

The better way turned out to be the Ensoniq 5503 Digital Oscillator Chip (DOC) included with the Apple IIGS. This is the same chip that appears in many of Ensoniq's synthesizers and MIDI (Musical Instrument Digital Interface) equipment.

The DOC contains 32 oscillators. These are paired to form 15 voices, each capable of producing its own sound (like 15 separate instruments in a band). The sixteenth voice is used internally for timing purposes.

Also included with the DOC is 64K of RAM referred to as *sound memory*, or *sound RAM*. Into this special area of memory can be placed various waveform patterns or even digitized samples of analog sounds such as a human voice.

The DOC can be programmed on two levels. Low-level programming involves reading and writing to the DOC's sound RAM and altering its registers directly. This method is complex, yet it's proven. In fact, the majority of the Apple IIGS sound applications
proven. In fact, the majority of the Apple IIGS sound applications
available use this technique. The second way to program the DOC is using the Apple IIGS Toolbox. This is the preferred way. The advantage of Toolbox routines becomes clear when you consider that three lines of code are required to play a note using the Toolbox and 30 or more lines of code and data statements would be required to play the same note using low-level routines. But there is a problem: The Toolbox sound routines aren't finished.

Soon, you'll be able to choose from a variety of sounds and tones as easily as opening a window. Apple is fast at work completing the sound routines. Unfortunately, they won't be finished in time for inclusion in this book.

The sound lab in the Advanced Technologies section of Apple Computer is impressive. The goal of the researchers is to create a sound environment for computers that is as advanced as the computer's graphics capabilities. While graphics have continually progressed, and programming the graphics has become easier, sound continues to be an orphan.

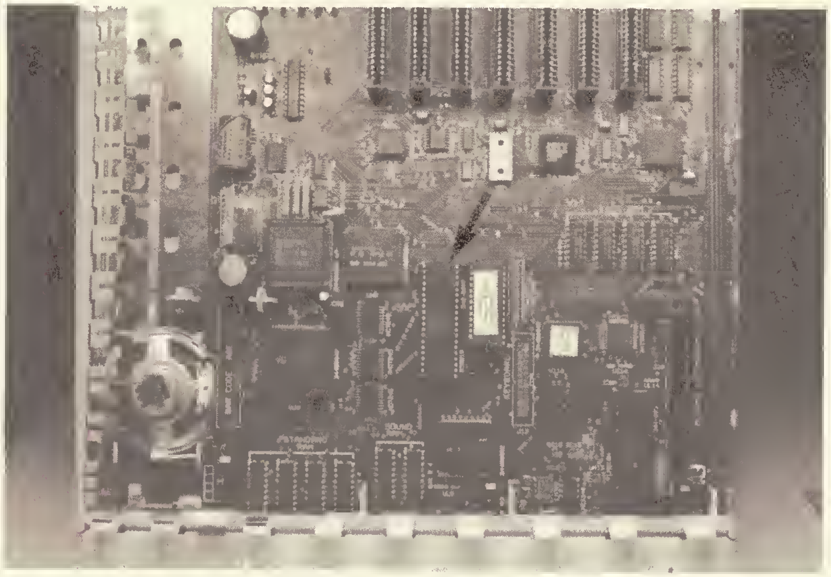
In the lab, they're concentrating on not only making sound easier to program, but on how to tailor sound toward specific applications. According to one of the researchers, "Any computer can go 'beep.' What about other sounds? How can they enhance the performance of a piece of software? How can sound help a user better interact with a program?"

Sadly, all this technological magic is being worked out on a Macintosh II, not an Apple IIGS. The researchers want you to know, however, that all information discovered will be shared with the IIGS development team. You may see interesting and exciting sound advancements on this computer in the near future.

The 65816 Processor

The actual brain of the Apple IIGS is the 65816 microprocessor. It's the latest generation of the 6502 series of processors. This family began with the 6502 microprocessor used in the first Apple computer. Since that time, the chip has been improved upon. It became faster and able to address megabytes of memory and handle 16-bit-wide operations.

Figure 2-1. Apple IIGS Motherboard with 65816 Pointed Out

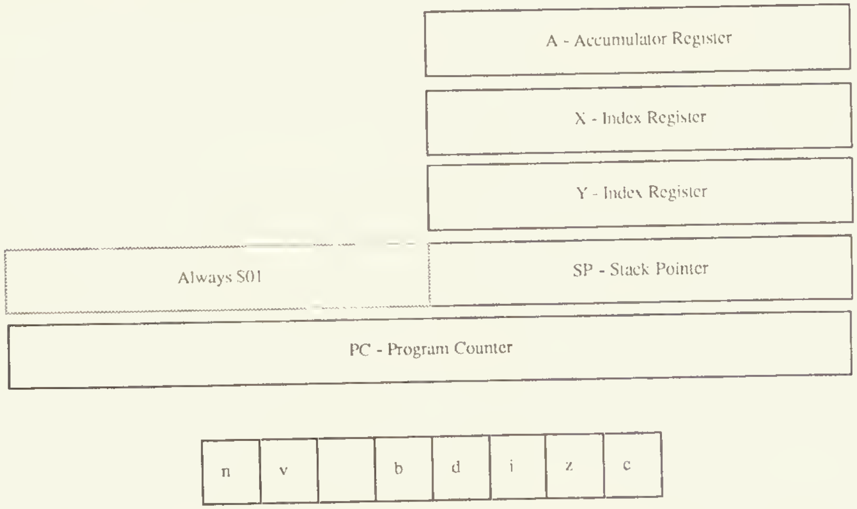


The 65816 is the brain of the Apple IIGS.

To maintain compatibility with the older 6502 chips (and the software that ran on them), the 65816 can emulate a 6502. In the emulation mode, it behaves exactly as a 6502 would, with very few exceptions. While emulating its ancestor, the Apple IIGS works on eight bits of data at a time and can access only 64K of memory.

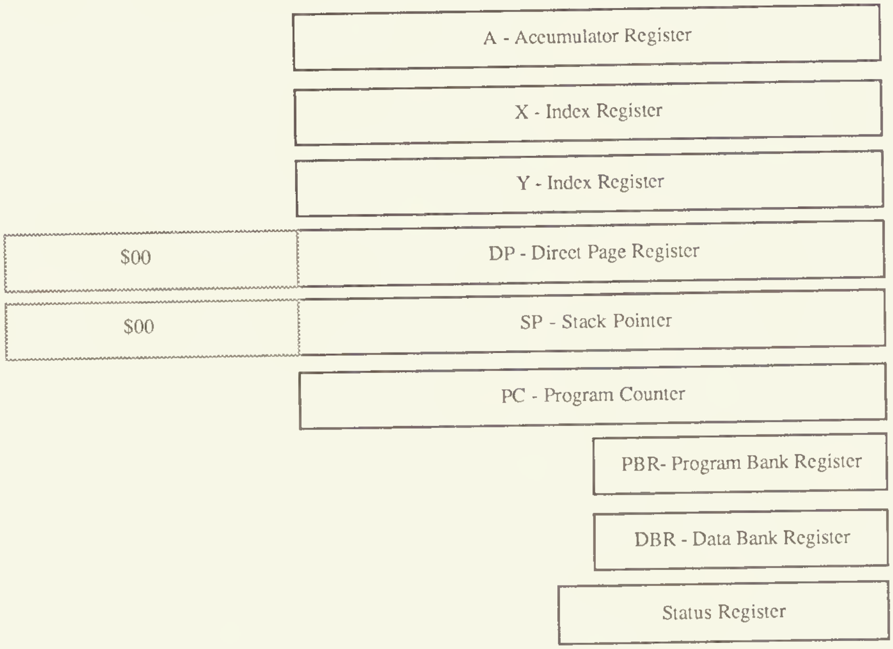
Note that while the 65816 is capable of emulating the 6502, older machines using the 6502 cannot run 65816 machine language programs. In fact, most of the 65816 machine language instructions are not defined for the 6502. Running a 65816 program on one of those machines (which would be hard to do in the first place) would crash the computer.

Figure 2-2. Diagram of 6502


A - Accumulator Register
X - Index Register
Y - Index Register
Always $01
SP - Stack Pointer
PC - Program Counter
n
v
b
d
i
z
c

The 6502 chip used in older Apple II machines can only handle eight-bit operations.

Figure 2-3. Diagram of 65816


A - Accumulator Register
X - Index Register
Y - Index Register
$00
DP - Direct Page Register
$00
SP - Stack Pointer
PC - Program Counter
PBR- Program Bank Register
DBR - Data Bank Register
Status Register

The 65816 can handle 16-bit operations as well as emulate the 6502.

When programming, it's possible to switch emulation on and off, as well as configure the A, X, and Y registers to either 8 or 16 bits. In machine language, this is done manually by setting the 65816 to emulation or native mode and by setting or clearing the register configuration bits. If you use the *APW* assembler, special assembler directives must be used to ensure that all following code is interpreted properly for the emulation mode. (See the *APW* manual for details.)

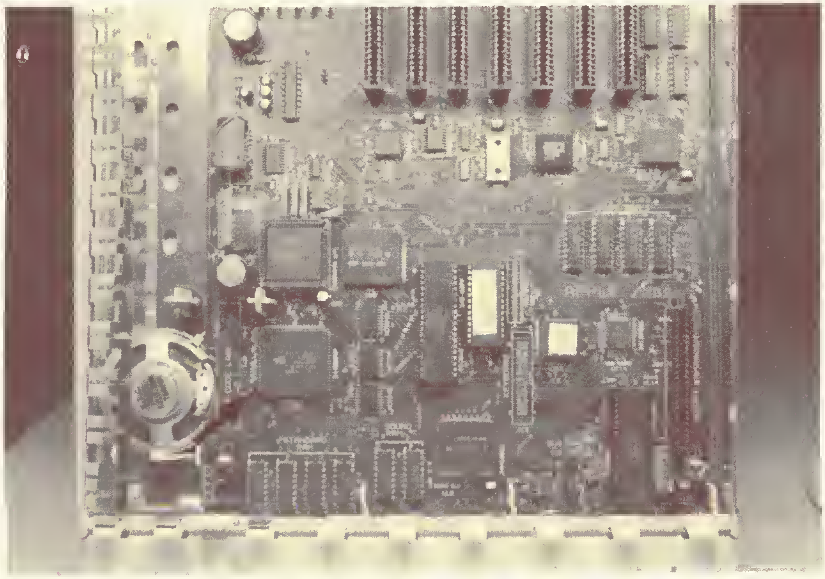
When programming in Pascal or C, emulation or native mode selection is taken care of automatically, either by default or through certain directives, depending on the software used. It's not necessary to ensure the processor is in one mode or the other when programming in Pascal or C.

To access the Toolbox, the 65816 must be in its native mode and all registers must be configured to 16 bits.

Apple IIe Emulation

One of the smartest things Apple Computer has done is to ensure that the software used on older Apple computers will work on new machines. Lack of compatibility has killed more than one microcomputer.

Figure 2-4. Apple IIGS Motherboard with Mega II Pointed Out



The Mega II: An Apple IIe all on one chip.

Programs that ran on the Apple II can run on the Apple II+. Apple II+ programs run on the Apple IIe and IIc. And the majority of those programs (about 90 percent) still run on a brand-new Apple IIGS. The reason for this is that the Apple IIGS contains a custom chip called the Mega II. The Mega II is an Apple IIe all on one chip.

Operation of the Mega II is transparent as far as programming the machine goes. While running older Apple II software, the Mega II takes charge and causes the machine to be an Apple IIe. When running Apple IIGS software, the Mega II does handle some operations. For the most part, however, its purpose is to emulate an Apple IIe and provide compatibility for older applications.

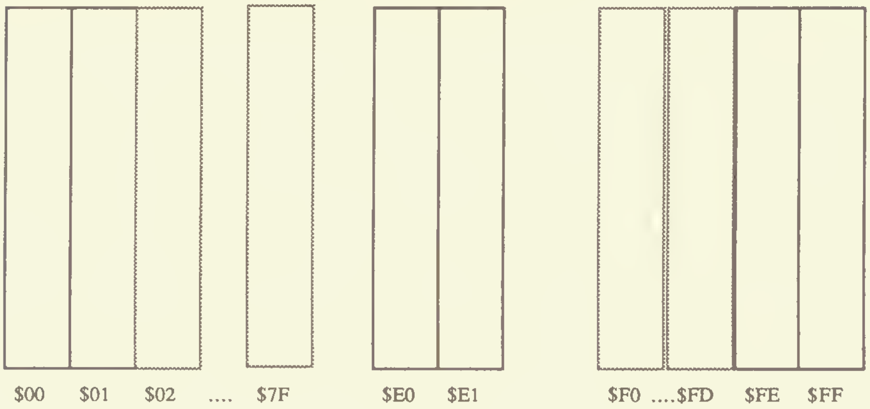
> The Mega II has an interesting history. Apparently, the Mega II took the Apple IIGS design team by surprise. People "upstairs" requested that the Mega II (supposedly designed for some other project) be used in the Apple IIGS. Because of this addition relatively late in the IIGS design, the Mega II chip contains many features made redundant by the VGC video chip.
>
> A slightly less-than-delighted design team did successfully incorporate the Mega II into the Apple IIGS, and it does perform its job very well. One question remains: What was the original purpose of the Mega II? A one-chip IIe or IIc, perhaps? Only time will tell.

Memory Addressing

The Apple IIGS has an alluring ability to address a tremendous amount of memory. This will be particularly attractive to programmers weaned on 64K (or even 128K) computers. Technically, the 65816 is capable of addressing 16 megabytes. The way the Apple IIGS is currently designed, only 8 megabytes of memory can be used for RAM, but that is still more than you're ever likely to need.

The eight megabytes of memory are divided into 128 separate banks of 64K each. The full 16 megabytes represents a total of 256 banks. Several of those banks are dedicated to the computer's ROM, possible ROM upgrades, and the Mega II chip. The memory map in Figure 2-5 shows how the memory banks are allocated in the Apple IIGS.

Figure 2-5. Memory Banks in the Apple IIGS



Each memory address (location) in the computer's RAM is represented by a bank number and an offset within that bank. For example, address $000200 indicates memory location $0200 in bank $00, the first bank of memory. Memory location $00A8 in bank $E1 is expressed as $E100A8. The first byte value represents the bank number; the second word value indicates an offset within that bank.

It's assumed that a leading $00 precedes all memory addresses. Because $E100A8 is not a true long-word value, the actual address is $00E100A8. However, because the MSB of the high-order word is always $00, it's usually left off (or assumed).

Allocating and controlling all this memory is the job of a very special tool set called the Memory Manager. One of the most important tool sets in the Toolbox, the Memory Manager is responsible for divvying up and setting priorities to blocks of memory. It's so well implemented that you need not know the exact location of a memory block. The Memory Manager takes care of all that for you. Blocks of memory can be moved, deactivated, or purged all via a call to the Memory Manager.

More details about memory and the Memory Manager can be found in Chapter 7.

Because the Apple IIGS currently comes only with 256K on the motherboard, you'll need to upgrade your machine's memory (as has been previously recommended). When you upgrade, you'll probably purchase a RAM card that allows you to use 256K RAM chips. Eight of these chips are equal to 256K of memory. The Apple IIGS considers 256K to be four banks.

As you add memory, the IIGS automatically assigns that memory to banks, beginning with bank $02. (Remember that you already have four banks of memory. Banks $00 and $01 are built-in FPI RAM, and the Mega II RAM and I/O are located in banks $E0 and $E1.) The typical memory card comes with at least four blocks which can each hold 256K of memory, making it capable of holding up to one megabyte of memory—16 banks of IIGS memory.

Memory cards with more than four blocks of 256K may cause some problems with future releases of the Apple IIGS. According to its designers, a memory card should have a maximum of four blocks of 256K. But certain hardware developers thought they could put more on a memory card. While the memory upgrade cards will still function, and the IIGS will be able to make use of the extra memory, some problems may result.

The best way to avoid problems when using a memory card with more than four blocks of 256K is to assign the extra memory as a ramdisk. This can be done using the Control Panel's ramdisk.

For example, the development systems this book was tested on contained RAM cards with 1.75 megabytes of RAM on them (six blocks of 256K). With an 800K ramdisk selected (the same size as the IIGS disk drive), the rest of the memory fit easily into the four-block maximum, and there were no problems.

Operating Environment

The operating system for the Apple IIGS is ProDOS 16, a custom operating system for the IIGS based on Apple's ProDOS 8 (which used to be just ProDOS). ProDOS 16 is very similar to ProDOS 8. In fact, updating is as easy as copying the ProDOS 16 files onto your old ProDOS 8 disks or hard drive.

ProDOS 16 controls disks and manages the file system. It uses the same file structure as ProDOS 8 and will even recognize, load, and run ProDOS 8 files, such as *AppleWorks*. However, ProDOS 8 cannot run the ProDOS 16 files. (And ProDOS 16 will not run on an Apple IIe, IIc, or II+.)

Incidentally, ProDOS 16 serves as a file-management system and isn't a true operating system in the sense that UNIX, MS-DOS, or OS/2 are operating systems. In fact, in the old days, all programming tasks were taken care of by the Apple's built-in BASIC interpreter. A program was run by typing its name at the BASIC command prompt, prefixed by a hyphen:

```
]-APLWORKS.SYSTEM
```

The above BASIC command would run the *AppleWorks* program, provided an *AppleWorks* disk was in the disk drive. (Another method to run *AppleWorks* was to place the *AppleWorks* disk into the primary disk drive and reboot the computer.)

The Apple IIGS provides a better way to interact with your programs.

Since late 1987, Apple introduced a Finder program, similar to the operating environment of the Macintosh. In fact, if you're familiar with the Mac, the IIGS Finder looks like a color version of the Mac's. Programs, data files, and file folders (which contain subdirectories) all appear as graphics images on the screen. The Finder allows files and programs to be manipulated with relative ease as compared to the older, slower ProDOS utilities. And, not only can ProDOS 16 and native Apple IIGS applications be run using the Finder, but because Apple also included a copy of ProDOS 8 on the Finder disk, older Apple II applications can be run as well.

Unlike the Macintosh's Finder, however, the Apple II Finder does have some limitations. Most notable among them is that not all Apple II applications are based on the graphic DeskTop environment. Older applications—and even some new ones—still use the Apple's text screen. Most of the newer applications, including examples in this book, will use the graphic environment of the DeskTop.

Chapter 3

How Programs Work

One trait most avid computer programmers share is a love of solving puzzles. Most great programmers are also great puzzle solvers. The self-taught computer wizard can unravel mysteries and evoke programming incantations that make a machine perform magical feats.

These programmers are not satisfied with just getting the job done. They want to make code tighter, faster, more ingenious. This chapter is directed to them.

This chapter explains how programs work on the Apple IIGS. Of course, if the subject doesn't make any sense, please read on. Whether you're a programming wizard or just an apprentice, this chapter contains interesting background information on how the IIGS works, how programs are loaded, what happens when they start, and where they go when they die. It's a chapter full of secrets revealed and undercover skullduggery—ideal for the potential programming prodigy.

Anticipation

Before you can begin serious programming on the Apple IIGS, you will need at least one disk drive and a system disk. The programming tips in this book were tested on one of the original computers using system disk version 3.1, as well as one of the later ROM 01 machines, so it should be applicable to your machine.

Of course, by the time you read this, ROM version 09 and system disk version 86 might be available. Things change that quickly. But don't worry. The information in this book is still good and all of it applies.

When you turn on your Apple IIGS, it looks for the *startup slot*. This is one of the slots on the motherboard into which a disk drive should be plugged. A specific startup slot can be specified in the Control Panel, or you can set it to *scan*. When set to scan, the Apple IIGS will scan all slots for the appropriate startup device.

When scan is selected, the system begins looking for an I/O device starting with slot 7 and continues searching down to slot 1. For example, if you have a hard disk drive connected to slot 7, that will be the startup device. Otherwise, the scan continues with slot 6 (the old floppy drive slot), slot 5 (the 3½-inch drive slot), and so on.

If you have selected a specific slot from the Control Panel, your IIGS will look for a startup device in that slot only. This way, if you had a disk controller card in slot 6 and you wanted the computer to startup from that device, it would do so, regardless of what was in the other slots or what devices were plugged into the IIGS ports (on the back panel).

The connectors on the back panel of the computer are really considered devices plugged into slots. In fact, if you run an old Apple IIe diagnostic program, it assumes you have every slot in the computer filled with specific devices, even though your IIGS may be totally empty inside.

Once the computer is turned on, its primary job is to find a disk drive. Once the disk drive is found, the computer checks to see whether a disk is in that I/O device. If not, or if the disk is of alien origin, the following message is displayed along with the Apple character bouncing back and forth across the screen:

```
Check startup device!
```

If a disk is found, the computer checks to see whether it's a boot disk, specifically, a ProDOS disk. If it's either a ProDOS 8 or 16 system disk, the system continues to load ProDOS from disk. If the disk is just a data disk (meaning there's no operating system present) the following is displayed:

```
*** UNABLE TO LOAD PRODOS ***
```

If this or the previous message is displayed, you should insert a ProDOS system disk into your disk drive and try again.

If you do have a bona fide ProDOS disk in the drive, your IIGS will attempt to load ProDOS into memory. For ProDOS 8, this is a very simple operation. For ProDOS 16, things are a little more complex.

The original disk operating system for the Apple II computer was simply called DOS, for Disk Operating System. It went through various iterations until its final version, DOS 3.3, was replaced by ProDOS in early 1983.

ProDOS was modeled after the SOS operating system Apple developed for the late Apple III computer. SOS stood for Sophisticated Operating System.

SOS introduced the hierarchical file system of volumes and prefixes now used by ProDOS. In fact, SOS files and ProDOS 16 files have identical structures to a certain extent. And because the Apple III Pascal used a file system similar to SOS, ProDOS 16 can read Apple III Pascal files as well.

Booting ProDOS 8

Because the Apple IIGS is Apple IIe compatible (for about 90 percent of the programs, according to the literature), it can load and run a ProDOS 8 program just as if it were a IIe. Due to this compatibility, it's logical to assume that both ProDOS 8 and ProDOS 16 are initially loaded from disk in a similar manner.

The program (actually ROM code) that loads ProDOS into memory is called Boot ROM. These instructions are located on the disk's controller card. The actual memory location of the Boot ROM is in memory bank $00, at location $C000 plus $100 times the slot number. So, if slot 6 contains the disk's controller card, the Boot ROM is at memory location $C000 plus $100 × 6, or $C600. (All memory locations from here on are in bank $00 unless otherwise specified.)

Boot ROM has only one job: to read in the first one or two blocks of the disk (or hard disk) into memory. The contents of these blocks are copied to memory location $800. With its dying breath, the Boot ROM's last job is to perform a JMP instruction to the machine language routines (loaded from disk) at the address $801.

The routine loaded from disk is $200 bytes long and occupies memory locations $800 through $9FF. If the disk being booted is formatted for ProDOS (either version), the information loaded from disk is called the ProDOS Boot Loader. This code will read in the rest of block 0, as well as the entire contents of block 1 of the disk. However, the information on block 1 is used primarily by the Apple III computer as a means of booting into the SOS operating system.

A block, the smallest unit of storage on a ProDOS disk, consists of 512 bytes of information. A sector, the smallest accessible unit of a DOS 3.3 formatted disk, holds only 256 bytes.

The Boot Loader's job, like the Boot ROM, is to load more information from disk—in this case, the rest of ProDOS. The ProDOS Loader searches the disk's *volume*, or *root*, directory for the file called PRODOS, which contains the ProDOS Relocator. If this file cannot be found, the following message is displayed:

```
*** UNABLE TO LOAD PRODOS ***
```

It's not unusual to see this when booting data disks. They're formatted for use with ProDOS, but aren't meant to be booted.

If the PRODOS file is found, it's loaded into memory locations \$2000–\$5BFF. And, like the Boot ROM, with the Loader's dying breath, it jumps to the machine language routines at address \$2000 which make up the ProDOS Kernel Relocator.

The ProDOS Relocator is the program that prints the ProDOS version number and copyright on the screen. It does a number of other interesting things: evaluating RAM, determining the type of Apple computer you have, and so on. But its biggest job is to copy the ProDOS Kernel, the actual operating-system part of ProDOS, to high memory. It also sets up the System Global Page. Incidentally, when the Relocator is copying the Kernel image to high memory, it makes a grating sound on the computer's speaker.

Once the relocated Kernel is running, ProDOS 8 scans the volume directory of the disk for a system file with a .SYSTEM suffix. If a .SYSTEM file is found—BASIC.SYSTEM, for example—it's loaded into memory at location \$2000, and then a JMP instruction is performed to that address.

If the .SYSTEM program is in fact BASIC.SYSTEM, the BASIC interpreter looks for a BASIC program named STARTUP in the volume directory. If found, that program is loaded into memory, and its instructions are executed.

This may seem like a very complex way of loading in something as simple as a BASIC program. Yet, nearly all microcomputers operate this way: First, a small bit of the disk is read, then a larger piece, and then, finally, the operating system is loaded into memory. It would probably be much more efficient to directly load the entire operating system when a computer starts, but not as flexible. Imagine all your data disks needing a 30-block boot sector simply to display the message, ****UNABLE TO LOAD PRODOS****.

Actually, a better justification for loading ProDOS in pieces is to allow the system to run more than one operating system. For example, using this method, an alien operating system could have its own Boot Loader on the first two sectors of a disk. This custom Boot Loader could then look for a special Loader file on disk—something other than PRODOS. The new Loader file could then load itself into memory and the Apple IIGS would run a new operating system, such as the old Apple Pascal.

You'll really appreciate the speed with which ProDOS 8 loads, especially after you have encountered the apparently sluggish ProDOS 16.

Booting ProDOS 16

As they're started, ProDOS 8 and ProDOS 16 are remarkably similar. They have to be similar, so they are compatible and use the same disk structure. But bear in mind that although the Apple IIGS can boot ProDOS 8 disks and run ProDOS 8 applications, older Apple IIs cannot run ProDOS 16 nor can they run Apple IIGS applications.

> Actually, as far as the computer is concerned, it doesn't matter whether the operating system is on disk or not. All it's looking for are the first two sectors, which it copies from disk into memory beginning at location $800. It then executes the instructions beginning at location $801, whether they mean something or not.

As with ProDOS 8, the first thing the Boot ROM does is load disk blocks 0 and 1 into memory location $800 in bank $00. The next step is also similar. In starting a ProDOS 16 disk, the program at $800 (the boot code) looks for a file named PRODOS in the volume directory—the same name as the ProDOS 8 Relocator. If the PRODOS file is not found, the ****UNABLE TO LOAD PRODOS**** message appears.

These similarities are not remarkable coincidences. This is because a disk formatted for ProDOS 16 will contain exactly the same boot code on blocks 0 and 1 as does a ProDOS 8 disk.

Once the jump is made to memory location $2000 (the PRODOS program), the two operating systems behave quite differently. The PRODOS program under ProDOS 8 is the Relocator and Kernel—the actual operating system. Under ProDOS 16, the PRODOS file loaded at memory location $2000 is just another link in a long chain of commands.

> If you try to boot ProDOS 16 on an Apple II other than a IIGS, the following is displayed:
>
> ```
> PRODOS 16 REQUIRES APPLE IIGS HARDWARE
> ```

The primary duty of the PRODOS file is to pass execution to the Apple IIGS System Loader. But before it does that, it sets up the ProDOS 16 quit code by transferring that part of itself to memory location $D000 in bank-switched memory. This code, referred to as PQUIT, stays in memory permanently and is used when a program quits. (The actual memory location is one of those pieces of information that you don't really need to know. There is no practical purpose for knowing that the code is loaded into the $D000 location, except to impress your friends.)

The Apple IIGS System Loader file is named P16. It's found in the SYSTEM subdirectory on the boot disk. The System Loader works closely with ProDOS as well as the Memory Manager to allocate, relocate, load, and save information between the disk drives and memory. As the System Loader (P16) is started, it displays a name and version number on the screen, just as ProDOS 8 does. See Figure 3-1.

Figure 3-1. System Loader Display

```
                    APPLE II

PRODOS 16 V1.3                          29-JUN-87

                   LOADER V1.3

     COPYRIGHT APPLE COMPUTER, INC., 1983-87
              ALL RIGHTS RESERVED.
```

The ProDOS version number appears after booting a system disk. V1.3 is the version and release number of ProDOS 16 (P16), as well as the System Loader (PRODOS) file. In the above example, both numbers are the same, though that may not always be the case.

Once ProDOS 16 is in memory, the PRODOS Loader (still in memory at $2000) continues its job. It looks in the SYSTEM/SYSTEM.SETUP subdirectory. All files in this directory are executed, starting with the file named TOOL.SETUP.

TOOL.SETUP patches or modifies any of the ROM tool sets (ID numbers $01–$0D). This file must be in the SYSTEM/SYSTEM.SETUP directory, and it is executed ahead of any other files in the subdirectory.

The SYSTEM.SETUP directory contains any file or program that needs to be loaded or initialized when the system is started. Primarily, two types of files can be included in SYSTEM.SETUP, along with TOOL.SETUP: Permanent Initialization files and Temporary Initialization files.

Permanent Initialization files. Permanent Initialization files have a file type of $B6. They're referred to as STR (STaRtup) files. These files are loaded and executed but not shut down like standard applications. They're actually more like subroutines because they're always in memory and end with an RTL instruction rather than calling the ProDOS Quit command. Permanent Initialization files must also be loaded into nonspecial memory and cannot allocate any stack or direct-page space.

An example of a Permanent Initialization file is the TOOL.SETUP program that patches the ROM-based tool sets. TOOL.SETUP contains adjustments and modifications to the ROM tool sets. It's actually an extension of the ROM code. When Apple learns of new bugs in the ROM tool sets, they release a new TOOL.SETUP file rather than new ROM chips. TOOL.SETUP must always be in memory, therefore it's a Permanent Initialization file and not a Temporary Initialization file.

Temporary Initialization files. Temporary Initialization files have a file type of $B7. They're referred to as TSF (Temporary Startup File) files. These files are similar to Permanent Initialization files, except they are shut down when completed, and their memory space is released. But, like Permanent Initialization files, they also end with an RTL instruction rather than calling the Quit function.

An example of a Temporary Initialization file is the BEEP.SETUP program listed later in this book. BEEP.SETUP replaces the normal system beep sound with a more pleasant noise. Once BEEP.SETUP completes its task, it's removed from memory (see Chapter 12 for more information on BEEP.SETUP).

After the SYSTEM.SETUP directory is scoured, and the STR and TSF programs are run, ProDOS looks in the directory SYSTEM/DESK.ACCS to load any desk accessories found there. Classic desk accessories (CDAs), with a file type of $B8, are placed into memory and can be accessed via the Control Panel. New desk accessories, with a file type of $B9, can only be used by DeskTop applications. All desk accessories in the SYSTEM/DESK.ACCS directory are loaded at this time.

You don't need to keep all your desk accessories in the SYSTEM/DESK.ACCS subdirectory—only those you want to load. Other desk accessories can be kept in a backup directory and then transferred to SYSTEM/DESK.ACCS for use when the system is rebooted.

After the desk accessories are loaded, ProDOS looks for a file named START in the SYSTEM directory. The file could be an applications file, or it could be the Finder or Launcher (discussed later). If a START file isn't found, ProDOS looks in the volume directory for a file with a suffix of either .SYS16 or .SYSTEM. The .SYS16 suffix indicates a ProDOS 16 file, and that program is loaded and executed. The .SYSTEM suffix is for a ProDOS 8 program.

If the .SYS16 program is found first, ProDOS calls its own quit code with the name of the .SYS16 file and executes it. If a .SYSTEM (ProDOS 8) file is found first, ProDOS calls a modified ProDOS 8 Quit call and executes the .SYSTEM file. However, in order to do this, the ProDOS 8 operating system file, P8 must be the SYSTEM directory.

If a SYSTEM/START file—or a .SYSTEM or .SYS16 file in the volume directory—does not exist, a fatal error occurs.

All this is done simply to boot the ProDOS 16 disk, so it's easy to see how ProDOS 16 can be accused of booting slowly when compared with ProDOS 8. However, given the power of this operating system and all the things it enables a programmer to do, it is well worth the extra wait.

ProDOS 16 Disk Contents

There are so many files on the ProDOS 16 boot disk that, even in the minimum configuration, all of them wouldn't fit on one of the old-style 140K disks. Most of these files and their duties were discussed in the previous section, but for review (and as a handy reference), they are touched on briefly here. The following programs (in alphabetic order) are on a sample system disk named /A/. Remember that throughout this section the volume name /A/ is used only for reference. Your system disk may have a different name.

/A/BASIC.SYSTEM The ProDOS 8 version of the BASIC interpreter. It contains the disk extensions to Applesoft BASIC in ROM.

/A/PRODOS The System Loader that is responsible for setting up the operating system, Toolbox, desk accessories, and generally getting the Apple IIGS running. Remember that both ProDOS 8 and 16 use the name PRODOS for their System Loader. One way to tell the difference is by looking at the file's size. ProDOS 8 is approximately 32 blocks in size, whereas ProDOS 16 is significantly larger at approximately 42 blocks. The sizes may vary depending on the release version, but ProDOS 16 will always be larger.

/A/SYSTEM/ The directory containing important files and folders (other directories).

/A/SYSTEM/DESK.ACCS Contains new and classic desk accessories to be loaded when ProDOS 16 boots. Other desk accessories can be included on your boot disk, but they will be loaded only if they are in this directory.

/A/SYSTEM/LIBS A directory holding system libraries. It appeared on the original System Disk, but not on the current (3.1) version. Apple may include it on future versions if an application needs library files.

/A/SYSTEM/P8 The ProDOS 8 operating system. If this file is renamed PRODOS and copied to the volume directory, the disk will boot as a ProDOS 8 disk.

/A/SYSTEM/P16 The ProDOS 16 operating system and Apple IIGS System Loader.

/A/SYSTEM/START A program to be run after ProDOS has finished loading (the startup program). It may be an actual application or a loader file to launch an application.

/A/SYSTEM/SYSTEM.SETUP A directory containing initialization files to be run at boot time.

/A/SYSTEM/SYSTEM.SETUP/TOOL.SETUP A required file used to patch tool sets in ROM.

/A/SYSTEM/TOOLS A directory containing all the disk-based tools for the Toolbox. The tool sets appear with the name TOOL followed by the three-digit decimal number of the tool set. So the Window Manager, tool set ID# $0E, appears in this directory as TOOL014.

The above are all the files of a typical ProDOS 16 system disk. Of course, more files exist depending on the application and version of the system disk. Besides DESK.ACCS and SYSTEM.SETUP in the SYSTEM directory, the following folders might also be found:

/A/SYSTEM/DRIVERS A directory containing control files for printers, *AppleTalk*, modems, and a variety of devices.

/A/SYSTEM/FONTS A directory containing a variety of fonts to be taken advantage of by programs that use them. The files are named after the font they describe, followed by a dot and the point size of the font. So, COURIER.10 is a ten-point Courier font, and TIMES.12 is a 12-point Times Roman font.

Two other files you might find on your system disk are these:

/A/SYSTEM/FINDER A program, run by the /A/SYSTEM/START program, that contains a DeskTop environment similar to the one found on the Macintosh.

/A/SYSTEM/LAUNCHER A simple program launcher, run by the /A/SYSTEM/START program. This program was around when the original Apple IIGS arrived and the Finder was not yet completed.

The Finder uses a number of other files on disk, most notably icon files containing the graphic images it uses as icons. The two icon files used by the Finder are DIALOG.ICONS in the volume directory and FINDER.ICONS in the directory /A/ICONS. (It's permissible to move the DIALOG.ICONS file into the ICONS subdirectory to keep your volume directory clean, by the way.)

If you're writing applications for distribution, you'll have to find a way to get the following files and programs on your ProDOS 16 disk:

/A/PRODOS
/A/SYSTEM
/A/SYSTEM/P16
/A/SYSTEM/SYSTEM.SETUP
/A/SYSTEM/SYSTEM.SETUP/TOOL.SETUP

These files are required by ProDOS 16 in order to boot successfully. However, the startup application will probably require tool sets and other support files.

For example, DeskTop programs may need /A/SYSTEM/FONTS/ (and the fonts) or /A/SYSTEM/START or the .SYS16 file in the volume directory. BASIC programs will need BASIC.SYSTEM and P8. And, if your program uses a disk-based tool set, you'll need to include it in the /A/SYSTEM/TOOLS directory.

If you plan to write and distribute your applications on a ProDOS 16 disk, you should know that your system disk and its contents contain software copyrighted by Apple. Only very wealthy companies can afford to pay the fees required to distribute ProDOS with their programs. For you, as a software wizard, it's best to put your applications on a data disk and then provide instructions for copying your software to a ProDOS disk or to have the user copy ProDOS and the Finder to your disk.

Contact Apple Computer for more information on licensing.

Launching Applications

Launching a program on the Apple IIGS is different from running programs on older Apples. The Apple IIGS offers a very diverse environment and, as usual, there are always a few more things going on than meets the eye. Of course, programmers will love to take advantage of the new features of ProDOS 16.

Because this book is about the Apple IIGS, ProDOS 8 is beyond its scope. There are many worthy texts already available on the subject to which the reader is referred. The concentration here will be on launching (or running) applications under ProDOS 16.

The first and most bizarre feature of ProDOS 16 is that programs start with a call to the ProDOS Quit function. A program starts by quitting.

To launch a ProDOS 16 application, the program can be one of three types:

- The program named START in the SYSTEM directory
- Any program with an S16 ($B3) file type
- Any program with a SYS ($FF) file type

The SYS file type is a ProDOS 8 application. Even so, the ProDOS 16 Loader will recognize this and, as part of the application's startup, ProDOS 8 will be loaded and executed, allowing you to run your ProDOS 8 program.

Programs can also be launched via the Finder or the Launcher. Whichever method is used, the program is loaded into memory and control is transferred to that program.

But there's considerably more to the story than that. If you have purchased this book and have read this far, you're probably interested in knowing the real information on program launching.

Launching. Your programs are actually loaded via the ProDOS 16 Quit call. When one application quits and performs the obligatory call to ProDOS notifying the operating system that it is finished, the program has the option of immediately running another program.

If a second program is not specified, the ProDOS 16 Quit call allows any previously launched programs to be rerun, either by reloading them from disk or restarting them from memory.

When the ProDOS 16 Quit function is called, the program making the call is basically finished. It can, however, tell ProDOS the following:

- Which program to run next
- Whether it can be used again after the next program quits

If the program doesn't specify the next program, ProDOS checks to see whether it can return to any other programs previously run, and if not, it executes the special quit code, PQUIT.

The ProDOS 16 Quit Function

Programming for ProDOS 16 is different than programming for the Toolbox, yet very similar to ProDOS 8. More information on using ProDOS is presented in Chapter 14. The ProDOS 16 Quit function is number $29. It has two parameters:

- The pathname of an optional program to run
- The quit-parameter word

To call ProDOS on the Apple IIGS, a long jump is made to the ProDOS vector in memory bank $E1, offset $A8:

```
jsl   $E100A8     ;Call the ProDOS vector
```

The JSL instruction is followed by two values. The first value is the function number, and the second is the long address of the parameter list:

Value	**Size**
Function number	Word
Parameter address	Long word

The function number is a word-sized value, and the parameter address is the memory location of a list of parameters required by the call. A sample Quit call in machine language would be

```
jsl     $E100A8        ;ProDOS vector
dc      i2'$29'        ;Function number $29, Quit
dc      i4'Params'     ;Address of Parameters
```

Or, if using macros (discussed in Chapter 4):

```
_QUIT   Params         ;see above
```

The information at the address indicated by the label Params contains the address of a pathname of a program to run, plus the quit parameter word. For example:

```
Params  Anop                ;Memory Address of parameters
        dc       i4'0'      ;A long word of zero, no pathname
        dc       i2'0'      ;quit parameter word of zero
```

This example would be used if a program were just quitting and not running another program. Using a long word of 0 for the pathname tells ProDOS to quit without running another program.

If the program were quitting and running another program, the following parameters might be used:

```
Params  Anop
        dc       i4'Prog'     ;The address of Prog's pathname
        dc       i2'0'        ;quit parameter word of zero
```

The label Prog, in this case, is the address of the pathname of a program to run next:

```
Prog    dc       i1'14'               ;must start with a count byte
        dc       '/GAMES/MONSTER'     ;pathname to run
```

In ProDOS, pathnames are always preceded by a count byte denoting the length of the path, which follows immediately. If a program were to quit with the above Params, the program MONSTER on the GAMES volume would be run.

This is how one program can run another and how the Finder, Launcher, *APW* shell, *TML Pascal* environment, and a plethora of other shells and operating systems will load and execute programs. They'll all do it via the ProDOS 16 Quit call.

The Quit-Parameter Word

If you don't want to run another program, or if you want to run another program and then have control come back to the original program, that is where you need the quit-parameter word.

The quit-parameter word is part of the ProDOS 16 Quit function's parameter list (see above). Out of the 16 bits of this word, only two are used. The rest are labeled *forbidden* by Apple:

Figure 3-2. The Quit-Parameter Word

Status:	Used		Reserved—set to 0													
Quit:																
Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Bit 15. Bit 15 of the quit parameter controls the quitting program's User ID (discussed later in this book) and whether or not the program will restart after a second program quits. (Each program has its own, unique ID number.) Bit 14 determines if the program quitting can be restarted from memory or should be reloaded from disk.

To stop one program, start another, and then return to the original program requires some fancy footwork. To assist in this ballet, ProDOS maintains something called a *Quit Return Stack.* As each program quits, it has the option of placing its User ID (uniquely identifying that program) onto the Quit Return Stack.

Likewise, when a program quits, ProDOS checks the Quit Return Stack for a User ID. If found, the program identified by the User ID is run again. It's like magic.

If Bit 15 of the quit parameter is set to 1, the quitting program's ID number is pushed to the ProDOS 16 Quit Return Stack. This means that, once a second program is done, control will return to the original program.

This is how programs like the Finder and Launcher work. When you select a program to run, the Finder sets bit 15 of the quit-parameter word and calls the ProDOS Quit function to run that program. Because this bit is set, the Finder or Launcher's User ID is saved on the ProDOS 16 Quit Return Stack. When the program you've selected is finished, ProDOS checks the Quit Return Stack, removes previous program's ID number, and returns to that program.

If Bit 15 were not set when the first program quits, then whatever program belongs to the User ID pulled from the Quit Return Stack is run. If the Quit Return Stack is empty, control returns to the PQUIT code established by PRODOS when the machine was booted.

Bit 14. Bit 14 of the quit-parameter word determines whether or not the program making the Quit call can be restarted from memory or should be reloaded from disk. If bit 14 is set to 1, the program can be restarted from where it sits in memory. If it is reset to 0, the program must be reloaded into memory by the System Loader. (This is all done by ProDOS. All you do is set or reset the bit.)

So, launching a program on the Apple IIGS starts with a Quit call. Quitting programs can specify the name of another program to run, as well as determine whether control returns to the original program after the second is run.

Programs may crash when run through a debugger because of the way the ProDOS 16 Quit function works in conjunction with the Quit Return Stack: When your program makes a ProDOS Quit call, the operating system becomes confused because the debug program is still running. This causes the system to crash. When using the trace mode in DEBUG, place a breakpoint before your code to make the ProDOS 16 Quit function call.

Computer States at Runtime

When ProDOS passes control to a program via the Quit call, the System Loader determines whether the new program is relocatable, or must reside at a specific location in memory. When this determination is done, the program is allocated its own space, given its own zero page, and enough memory to operate. A number of other things can happen, depending on the program and how it was loaded.

Only file types $B3–$BE can be loaded by the System Loader, and only file types $B3 and $B5 can be run as programs (and specified by a Quit call). If a file of an unusual type is specified, the System Loader reports error $5C, *Not an executable file.*

Table 3-1. ProDOS 16 Load File Types

Type	Hex	Dec	Description
S16	B3	179	ProDOS 16 system application file
RTL	B4	180	*APW* runtime library file
EXE	B5	181	ProDOS 16 shell application file
STR	B6	182	ProDOS 16 Permanent Initialization File
TSF	B7	183	ProDOS 16 Temporary Initialization File
NDA	B8	184	New desk accessory
CDA	B9	185	Classic desk accessory
TOL	BA	186	ProDOS 16 tool set file
DRV	BB	187	ProDOS 16 driver file
...	BC	188	System use
...	BD	189	System use
...	BE	190	System use

Unlike older ProDOS 8 applications, there is no way to be certain exactly where a program running under ProDOS 16 will be put in memory. (ProDOS 8 programs were always loaded at memory location $2000 in bank $00.) However, there are a few guarantees made by Apple regarding the state of the system when your program takes control.

As with the Boot ROM, once the Loader places your program into memory, control of the machine passes to the first instruction of your program. Because the Apple IIGS is a single-tasking computer, meaning it's capable of doing only one thing at a time, your program has complete control when it starts. The computer states listed in Table 3-2 will be set at the time your application is launched.

Table 3-2. The 65816 Registers Set at Launch

Register	Type	Value
A	Accumulator	The application's User ID
X	Index	$0000
Y	Index	$0000
S	Stack pointer	The top of stack space
D	Direct page	The bottom of stack space
P	Processor status	All zero, native 65816 mode
PBR	Program bank	Determined by the Loader
DBR	Data bank	Determined by the Loader
PC	Program counter	Determined by the Loader

The addresses pointed to by the S and D registers are in bank $00. (The stack and direct page must always be in bank $00.) For example, the S register might point to $1BFF, and the D register might point to $1800, defining the stack and direct-page space to that $400 byte block. Note, however, that tool sets must request their own direct-page space from the Memory Manager (see the next chapter).

The values of the program and data bank registers, as well as the program counter will be determined by the Loader and what your application requires. There is no guarantee that the program-bank and data-bank registers will be pointing to the same bank of memory.

Other aspects of the system are set as follows:

- The standard input and output devices used by the Text tool set are both set to the Pascal 80-column video screen. These can be changed by using the Text tool set commands to specify new input or output devices. However, at startup, both are set to the Pascal 80-column device, also loosely referred to as *the screen.*
- *Memory shadowing* is set on for the language card, I/O spaces, and text pages, and is set off for the graphics pages. Unless you are truly an expert, it is not recommended that you alter memory shadowing.

Chapter 4

About the Toolbox

The Toolbox is crucial to programming the Apple IIGS. All the routines necessary for programming the Apple IIGS are kept in the Toolbox. But the Toolbox is more than a simple set of programming routines: It's the secret to writing programs and developing DeskTop

applications for the Apple IIGS. Know the Toolbox, and you can master the machine.

This chapter introduces the Apple IIGS Toolbox. The Toolbox contains about 1000 unique routines (called *functions*) that take much of the effort out of programming the Apple IIGS. Though the name *Toolbox* is accurate when it describes these routines and functions as tools, it might be more fitting to refer to the Toolbox as a treasure chest of programming features.

This chapter won't detail the operation of the Toolbox, but it does show how to use the Toolbox to your best advantage. For detailed information about the Toolbox, including a complete list of the Toolbox function numbers and parameters, refer to a comprehensive reference, such as that found in *COMPUTE!'s Mastering the Apple IIGS Toolbox.*

Toolbox Briefing

The Toolbox contains routines found in the computer's ROM as well as some routines that must be loaded from disk into RAM (called *disk-based* tools). The nearly 1000 unique functions in the Toolbox are grouped into 28 different categories called *tool sets.* For example, all of the functions related to the manipulation of windows are found in the Window Manager tool set, the pull-down menu functions are in the Menu Manager tool set, and so on. (See Table 4-1 for a complete listing.)

A tool set can contain as many as 255 different functions. At present, the QuickDraw II tool set, the largest by far, contains 206 unique routines.

Each tool set function is given a unique identification number. The number shows which tool set the function belongs to and gives the individual function number within that tool set. Together, these two numbers create a two byte (16-bit, or word-sized) number identifying the function. One byte gives the tool set; the other, the function number:

function number (1 byte) tool set number (1 byte)

The byte representing the function number comes first, followed by the tool set. It's backwards, but it's consistent. All of the functions in the Toolbox are identified this way. For example, the Miscellaneous tool set is tool set number $03. A function within that tool set, SysBeep, is function number $2C. SysBeep is referred to as function $2C03 in the Toolbox:

function number ($2C), tool set number ($03)
SysBeep = $2C03

The tool set ID is the low-byte value of $03, and the function ID is the high-byte value of $2C. Any other function in the Miscellaneous tool set will also end with the low-byte value of $03, but it will have a different high-byte value.

Table 4-1 contains a complete list of tool sets, their names, and ID numbers. Note which ones are found in ROM and which ones are located on disk.

Table 4-1. Tool Set Chart

ID	Name	Where	Comments
$01	Tool Locator	ROM	
$02	Memory Manager	ROM	
$03	Miscellaneous tool set	ROM	
$04	QuickDraw II	ROM	$300 bytes direct-page space
$05	Desk Manager	ROM	
$06	Event Manager	ROM	$100 bytes direct-page space
$07	Scheduler	ROM	
$08	Sound Manager	ROM	$100 bytes direct-page space
$09	Apple DeskTop Bus	ROM	
$0A	SANE	ROM	$100 bytes direct-page space
$0B	Integer Math	ROM	
$0C	Text tool set	ROM	
$0D	RAM Disk	ROM	Internal use only
$0E	Window Manager	Disk	Uses Event Manager's direct page
$0F	Menu Manager	Disk	$100 bytes direct-page space
$10	Control Manager	Disk	$100 bytes direct-page space
$11	System Loader	Disk	
$12	QuickDraw II Auxiliary	Disk	Uses QuickDraw's direct pages
$13	Print Manager	Disk	$200 bytes direct-page space
$14	Line Edit	Disk	$100 bytes direct-page space
$15	Dialog Manager	Disk	Uses Control Manager's direct page
$16	Scrap Manager	Disk	
$17	Standard File	Disk	$100 bytes direct-page space
$18	Disk Utilities	Disk	(No information)
$19	Note Synthesizer	Disk	(No information)
$1A	Note Sequencer	Disk	(No information)
$1B	Font Manager	Disk	$100 bytes direct-page space
$1C	List Manager	Disk	

The tool set ID is the identification number used to reference the tool set during calls to functions. For the sake of convenience, and to be consistent with Apple's documentation, hexadecimal (base-16) notation is used. This also makes it easier to spot the tool set number when looking at only a two-byte Toolbox function value.

The names listed in the second column of Table 4-1 are the official tool set names. The purposes of most tool sets may be easily discerned from their names. The Miscellaneous tool set, number $03, contains a hodgepodge of important functions that don't fit comfortably under the rubric of any of the other tool sets.

The third column in Table 4-1 indicates whether a tool set is located in ROM (built into the IIGS) or whether it is loaded into RAM from disk.

Additional information is listed under Comments, such as how many direct pages are required by the tool set. The tool sets often need a certain amount of direct-page memory. Its use is similar to BASIC's use of zero page: as a scratch pad for temporary storage of data and pointers. The amount needed depends on the tool set, and its use is discussed in greater detail later in this chapter.

Opening the Toolbox

Before the Toolbox can be accessed, the microprocessor must be placed into native mode. That is, the computer must be running with Apple IIe (Mega II) emulation turned off. Additionally, all registers in the 65816 microprocessor must be set to 16-bit widths. The following code does this in machine language:

```
clc             ;clear the carry bit
xce             ;and the emulation bit
rep     #$30    ;use 16-bit memory and registers
```

Depending on where and how an application has been launched, the code above may not be necessary. If the *APW* or *ORCA/M* assembler is used, there's no need to establish the size of the registers and turn off emulation. However, with other assemblers and especially for BASIC programs using the Toolbox with machine language subroutines, you must perform the above operation. The Toolbox cannot be accessed when the 65816 microprocessor is in emulation mode.

With high-level-language compilers you don't need to worry about turning off emulation. All ProDOS 16 program launchers automatically set the 65816 into native (non-emulation) mode before your application starts.

Calling the Toolbox

To call the Toolbox using machine language, place the function ID (tool set number and function number) in the X register. Push onto the stack any parameters passed to the function. Finally, make a long jump to the subroutine (JSL) at address $E10000, the Toolbox dispatcher. Any parameters returned from the function should be pulled from the stack after returning from the function.

The first Toolbox commandment: Thou shalt not access a Toolbox function unless its tool set has been started up. Every function in the Toolbox is part of a specific tool set. And before that function can be used, its tool set must be started.

Each tool set has a special function to do this, called the StartUp function. This function is always function number $02. So, before you can use any function in the Miscellaneous tool set, you must call the MTStartUp function, ID number $0203 ($02 for the StartUp function and $03 for the Miscellaneous tool set). Once StartUp is called, other routines in the tool set can be accessed.

The specifics of calling the Toolbox, along with step-by-step analysis, is provided in *COMPUTE!'s Mastering the Apple IIGS Toolbox*. Refer to that text if these concepts are new to you.

To perform the MTStartUp function in machine language, the X register is loaded with the 16-bit function ID number, $0203 and then a JSL instruction is made to memory location $E10000. (JSL is Jump to Subroutine Long, and memory address $E10000 is the memory location of the Toolbox.) This is the door through which you get to the Toolbox.

So in order to start this tool set, a machine language program would use the following code:

```
ldx     #$0203    ;MTStartUp
jsl     $E10000   ;start the Miscellaneous tool set
```

The short form of this call is

```
_MTStartUp        ;Start the Miscellaneous tool set
```

This is an *APW* assembler macro call defined in the M16.MISCTOOL macro file (macros and macro files are discussed in the next chapter). Throughout the remainder of this book, both the long and short (macro) forms of making Toolbox calls in the assembler will be used.

In C, calling the StartUp function is as easy as typing the function name. For example, to start up the Miscellaneous tool set, the following is used:

```
MTStartUp( );
```

And it's done. The information needed by the compiler to perform the Toolbox call is contained in an include file. Just use the Toolbox function name in your source code, and the function is called automatically. Remember to place the following at the top of your C source code listing:

```
#include <misctool.h>
```

With Pascal, making a Toolbox call is just as easy. Using *TML Pascal*, the Miscellaneous tool set is started as follows:

```
MTStartUp;
```

As with C this is simply a statement in your Pascal source code. The information is built into the *TML Pascal* unit file called MISCTOOLS.USYM. In the USES portion of your Pascal program, you would include this file in the following manner:

```
USES    MiscTools;
```

Once the tool set has been started up, an application can use its features. For example, the SysBeep function, which beeps the speaker, is function number $2C03 of the Miscellaneous tool set. To call the system beep procedure in machine language, use the following:

```
ldx     #$2C03      ;the SysBeep function ID
jsl     $E10000     ;call the Toolbox
```

or, if using macros:

```
_SysBeep            ;call SysBeep
```

Remember, the StartUp function has already been called. For C, the source would be

```
SysBeep( );
```

and in Pascal, simply

```
SysBeep;
```

As mentioned previously, in Pascal each Toolbox function call contains its definition in a support file. These files can be included, used, or copied into your source file, depending on which language your program speaks. For example, with the *APW* assembler, the MCOPY command is used to copy macro definitions from external macro libraries into your program. In the C language, the #include directive causes the compiler to include a *header file* defining the Toolbox calls, as if it were an extension of your source code. Similarly, *TML Pascal* incorporates unit symbol files which are brought into the compilation step with the USES statement.

These techniques of including, using, or copying are all covered in the next chapter.

Tool Set Interdependencies

Many tool sets in the Toolbox call upon other tool sets to perform a special operation. This collaboration requires that interdependent tool sets—those that rely upon others—must be active and available.

While your program may only deal directly with the Menu Manager, the process of drawing menus relies upon the graphics wizardry of QuickDraw II. So your application must start up both the Menu Manager and QuickDraw II. To further complicate matters, the order in which the tool sets are started is equally important.

Fortunately, the following table presents a list of the interdependent tool sets, the tool sets they need, and the order in which they should be started:

ID	Tool Set	Tool Sets Required (by Tool Set ID)
$01	Tool Locator	None
$02	Memory Manager	$01
$09	DeskTop Bus	$01
$0B	Integer Math	$01
$0C	Text tool set	$01
$0A	SANE	$01, $02
$16	Scrap Manager	$01, $02

ID	Tool Set	Tool Sets Required (by Tool Set ID)
$03	Miscellaneous tool set	$01, $02, $0B
$04	QuickDraw II	$01–$03
$07	Scheduler	$01–$03
$08	Sound Manager	$01–$03
$19	Note Synthesizer	$01, $02, $08
$11	System Loader	$01–$03
$12	QuickDraw Auxiliary	$01–$04
$06	Event Manager	$01–$05, $09
$0E	Window Manager	$01–$06, $10, $0F
$14	Line Edit	$01–$04, $06, $16
$10	Control Manager	$01–$04, $06, $0E, $0F
$0F	Menu Manager	$01–$04, $06, $0E, $10
$1C	List Manager	$01–$04, $06, $0E, $10, $0F
$15	Dialog Manager	$01–$04, $06, $0E, $10, $0F, $14
$05	Desk Manager	$01–$04, $06, $0E, $10, $0F, $14, $15, $16
$17	Standard File	$01–$04, $06, $0E, $10, $0F, $14, $15
$1B	Font Manager	$01–$04, $0B, $0E, $10, $0F, $1C, $14, $15
$13	Print Manager	$01–$04, $12, $06, $0E, $10, $0F, $14, $15, $1C, $1B

For example, if your program uses any functions in the Line Edit tool set, it must start up in the following order: tool sets $01–$04 (Tool Locator, Miscellaneous tool set, Memory Manager, QuickDraw II), tool set $06 (Event Manager), and tool set $16 (Scrap Manager).

The First Six Functions

Consistency has never been highly regarded in the computer programming world. But the Apple IIGS programmer will be delighted to know that the first six function calls in each tool set follow a standard format. These functions are housekeeping, or tool set management routines, and every tool set has them.

As shown in the previous section, the StartUp function must be called before other functions in a tool set can be used. StartUp is just one of the first six functions.

Apple Computer has reserved tool set functions $07 and $08 for future enhancements. Until they are placed on the duty roster, the next usable function in each tool set is $09.

The first six function calls in each of the first six tool sets are as follows:

ID	Function	Description
$01	BootInit	Initializes the tool set for the first time
$02	StartUp	Starts up the tool set for application usage
$03	ShutDown	Shuts down the tool set when no longer needed
$04	Version	Returns the version number of the tool set
$05	Reset	Initializes the tool set after a system reset
$06	Status	Determines whether the tool set is active or not

These function names are unique to each tool set since they are always prefixed by a short name. For example, the Memory Manager uses the letters *MM* before each of these function names: MMStartUp, MMVersion, and so on. The Tool Locator tool set uses TL: TLBootInit, TLStartUp, and so on.

- BootInit must never be called by an application. If the tool set is ROM-based, this function is performed when the computer starts up. If the tool set is RAM-based (loaded from disk), BootInit is called after it is first loaded into memory.
- StartUp: As stated in the previous section, applications must call StartUp so that the tool set's functions become available. Some tool sets require input parameters for use with the StartUp function. Passing parameters to a Toolbox function is discussed in the next section.
- ShutDown must be called before exiting to the operating system when an application is finished with a tool set. The tool set would then free up any memory it had allocated and, in general, would clean up after itself.
- Version: An application can determine the version number of a tool set by calling this function. It returns a word (16-bit integer) result. The high-order byte of the result consists of the major version number. The low-order byte contains the minor version. If the tool set is a prototype, bit 15 of the version number result will be set. (In this text when a bit is said to be *set*, it is made equal to 1. A *reset* or *cleared* bit is one made equal to 0.)
- Reset occurs when you press Control-Reset or make a DeskTop Bus reset call from software. The computer performs the Reset function in each of the active tool sets.
- Status: A program can find out if a tool set has been started by making a call to it's Status function. If not active, it returns an integer value of 0, otherwise it returns a nonzero value.

Passing and Receiving Arguments from the Toolbox

The majority of the Toolbox functions require an argument (a value or parameter) to be sent to the Toolbox, or they return an argument, or a combination of both. The Toolbox works with three types of parameters: bytes, words (two bytes), and long words (two words).

If any arguments are required by a function, they are pushed onto the processor's stack before the Toolbox call is made. Arguments returned from a function are then pulled from the stack after the call. This is demonstrated in the following portion of code which obtains the version number of the Miscellaneous tool set:

```
pha                 ;push space for the result
ldx    #$0403       ;the MTVersion function
jsl    $E10000      ;call the Toolbox
pla                 ;retrieve version information
```

The values returned from the stack must have space reserved for them before the call is made. This is done by pushing arbitrary values onto the stack. These values are replaced with useful information, pulled from the stack, after the call is made.

The above function is handled as follows in C:

```
Version = MTVersion( );
```

Note that Version must be declared beforehand as a word value, an unsigned integer. After the call, the Version variable contains the version number of the Miscellaneous tool set.

In Pascal, the function call is similar:

```
Version = MTVersion;
```

Remember to declare the variable, *Version,* as an integer. In both Pascal and C, the code for stack manipulation is provided by the compiler.

When starting up the Memory Manager, a word-sized value is pushed onto the stack before the call to MMStartUp is made. This provides the result space for an ID number:

```
pha                 ;push space for the result
ldx    #$0202       ;MMStartUp
jsl    $E10000
pla                 ;pull the user ID
sta    UserID       ;save it in a safe place
```

When MMStartUp is called, not only does it allow access to other Memory Manager functions, it also assigns your application a unique identification number. You should store the value pulled from the stack as your program's User ID. You'll need it later on.

The Memory Manager is covered in detail in Chapter 7.

Direct Pages

Many tools need only a call to their StartUp function to get them going. Others require additional information, such as timing information, graphics modes, the User ID returned by the Memory Manager's StartUp function, or a combination of these.

A few tool sets require a small block of RAM to use as scratch space for their functions. This memory buffer is called a direct page, and it consists of one page (256 bytes) of RAM. The direct-page memory must exist in the first 64K bank of memory.

Space for the direct page is allocated using a function in the Memory Manager. This function is called NewHandle. Since a program may use many tool sets and require a large quantity of direct-page space, it's common to allocate one large block of memory for use by each of the tool sets requiring direct pages. Therefore, you should calculate the total amount of direct-page memory needed before using the NewHandle function. See Table 4-1 for the amount of direct-page space each tool set requires.

Once the direct page is established (by some sleight-of-hand programming you'll be reading about later), portions of it are divided among the tool sets which require them.

Tools on Disk

Some tool sets are stored in the SYSTEM/TOOLS subdirectory on the ProDOS 16 disk your computer is booted with. Tools on disk cannot be accessed until they have been loaded into memory. This is accomplished with the LoadTools function of the Tool Locator tool set.

LoadTools uses a list of tool set numbers in memory to load corresponding files from disk. It accesses the disk and copies the tools into memory.

When calling LoadTools, an application first pushes a four-byte address of the tool list to the stack. For example:

```
pea   Toolist|-16     ;push long word address of list
pea   Toolist
ldx   #$0E01          ;LoadTools
jsl   $E10000
```

Toolist (above) points to the memory location of the list of tool sets to be loaded from disk. The structure of the list of tool sets begins with a count word (two bytes) which tells LoadTools how many entries there are in the list.

The count word is followed by several four-byte entries that describe the tools to be loaded. The first two bytes constitute a word that contains the tool set's ID number. For example, $0003 would indicate the Miscellaneous tool set. The second two bytes are a word that specifies the minimum version of the tool. If a program requires version 1.3 or later of a tool set, $0103 is specified. By using a minimum version number of $0000, any version on disk will be loaded.

The following is a sample table showing three tool sets to be loaded from disk: the Window Manager (tool set $0E), the Menu Manager (tool set $0F), and the Control Manager (tool set $10).

```
Toolist  dc    i‘3’              ;count word (3 tool sets)
         dc    I‘$0E’,I‘0000’    ;Window Manager 0.0 or newer
         dc    I‘$0F’,I‘0000’    ;Menu Manager 0.0 or newer
         dc    I‘$10’,I‘0000’    ;Control Manager 0.0 or newer
```

After the LoadTools call is complete, the program can proceed by starting up each of the loaded tool sets as needed.

In C, the method of loading tools from disk starts by globally declaring an array of tool sets as a group of unsigned word-length integers:

```
Word   Toolist[ ] =    {3,         /* Tool count */
                       14, 0,      /* Window Manager */
                       15, 0,      /* Menu Manager */
                       16, 0};     /* Control Manager */
```

From within a function in your application, the LoadTools( ) function is called in this manner:

```
LoadTools(Toolist);
```

If you're using Pascal, the procedure is almost the same, except Toolist is defined in the VAR section of the program as a ToolTable type, a special record which follows the structure of the tool list:

```
Toolist: ToolTable;
```

Unfortunately, Pascal forces the values in the Toolist array to be assigned at run time within a procedure. This results in longer code. Example:

```
Toolist.NumTools := 3;              { Tool count }
Toolist.Tools[1].TSNum := 14;       { Window Manager }
Toolist.Tools[1].MinVersion := 0;
Toolist.Tools[2].TSNum := 15;       { Menu Manager }
Toolist.Tools[2].MinVersion := 0;
Toolist.Tools[3].TSNum := 16;       { Control Manager }
Toolist.Tools[3].MinVersion := 0;

LoadTools(Toolist);
```

However, the LoadTools function call is identical in syntax to the call in C.

When Errors Occur

Calling some Toolbox functions can result in errors. Errors can occur under a variety of circumstances. Not all of them are fatal.

The way to tell whether there was an error during your Toolbox call is to test the carry flag after the function returns. If the carry flag is set, an error occurred, and your program can take appropriate action. If the carry flag is clear, no error occurred, and the program can continue.

If an error does occur, the Toolbox places a special error code in the A register. This two-byte value describes the error that occurred and the tool set called. Unlike the Toolbox function numbers, the tool set number in an error code is in the upper byte. The error number is in the lower byte. For example, if the error returns $0110 in the A register, the upper byte ($01) indicates that the error occurred with tool set $01, the Tool Locator. The error code ($10) is in the lower byte. Error code $10 of the Tool Locator is *Minimum Version Not Found.* (All error codes are documented along with the Toolbox functions in *COMPUTE!'s Mastering the Apple IIGS Toolbox.*) This error might occur when the LoadTools function is called to load tool sets from disk into RAM. If the minimum version specified is not found on disk, this error is returned after the LoadTools function is called.

Note that only some of the functions in the Toolbox result in actual errors. Some are unable to produce errors, yet may return with the carry flag set. An application should only test for errors after making Toolbox calls capable of producing errors.

Trapping for Toolbox errors in a C program is done by testing an external variable called _toolErr (note the underscore). This variable is declared as type extern in the types.h header file, which should be the first file included by any C program that uses the Toolbox. If _toolErr is a nonzero value, it means that the most recent Toolbox function resulted in an error. The value in _toolErr is the error code.

Here is a sample error-handling statement in C:

```
if (_toolErr) SysFailMgr(_toolErr, nil);
```

Care should be taken when handling errors in C by referencing the _toolErr variable. Since this variable is changed after each function call, your program should make a copy of _toolErr before using any other Toolbox functions.

TML Pascal programmers handle errors in a similar fashion. To see if an error has occurred, the value of a predefined variable called IsToolError is tested. The error code is stored in another predefined variable called ToolErrorNum.

Here is a sample error-handling statement in *TML Pascal:*

```
IF IsToolError THEN
        SysFailMgr(ToolErrorNum, ‘Fatal system error —> $’);
```

All the examples for handling errors, shown here, take the easy way out. The Miscellaneous tool set includes a function called SysFailMgr which brings up the familiar sliding Apple error message screen. (You see it when you try to boot the Apple IIGS without a disk in the drive).

SysFailMgr is adequate for testing purposes, but it shouldn't be used when errors occur in end-user or commercial applications. There are elegant (and user-friendly) ways of handling errors. It just takes a little extra effort to incorporate them into your programs.

Closing the Toolbox

When an application is finished using a particular tool set, it should shut it down. This is done by calling the tool set's ShutDown function, number $03. For example, to shut down the Menu Manager, the MenuShutDown call is made:

```
ldx   #$030F      ;MenuShutDown
jsl   $E10000
```

Since an application uses many tool functions throughout the running of the program, tool sets are usually shut down all at once before the program quits.

As a rule, tool sets should be shut down in the reverse order that they were started up. If, for example, the Miscellaneous tool set was shut down before other tool sets, it would cause the application to crash.

The Memory Manager is one of the last two tool sets to be shut down just before a program ends. Before the MMShutDown call is made, all allocated memory handles associated with an application should be disposed (that is, those requested for direct-page space). The easiest way to do this is with the DisposeAll function:

```
lda   MemID       ;Identify the blocks
pha               ;. . . by their ID numbers
ldx   #$1102      ;DisposeAll (memory handles)
jsl   $E10000
```

This disposes of all memory handles allocated by the application (identified by the MemID value). DisposeAll should never be used with the UserID value that was returned by MMStartUp.

Handles, doled out by the Memory Manager's NewHandle function, can be disposed of one at a time. This example demonstrates how easily a handle can be removed from C or Pascal:

```
DisposeHandle(MyHandle);
```

When memory handles are disposed, the space they occupied is freed and is made available to other applications. More details on memory management are discussed in Chapter 7.

Once all the memory handles allocated by your program are disposed, the MMShutDown function can be called.

Chapter Summary

The following Toolbox functions were referenced in this chapter:

Function:	$0E01
Name:	LoadTools
	Loads a list of tools from disk into RAM
Push:	Tool List Address (L)
Pull:	nothing
Errors:	$0110 Version Error; possible ProDOS errors
Comments:	The list of tools starts with a count word.

Function: $0302
Name: MMShutDown
Shuts down the Memory Manager
Push: User ID (W)
Pull: nothing
Errors: none
Comments: The User ID is obtained when MMStartUp is first called.

Function: $1002
Name: DisposeHandle
Disposes of a handle and the memory block it references
Push: The Handle (L)
Pull: nothing
Errors: $0206 (invalid handle)

Function: $1102
Name: DisposeAll
Disposes of all memory handles associated with an ID
Push: User ID (W)
Pull: nothing
Errors: $0207 (invalid User ID)
Comments: Do not use with the program's master User ID.

Function: $0203
Name: MTStartUp
Starts up the Miscellaneous tool set
Push: nothing
Pull: nothing
Errors: none
Comments: This call must be made before any Miscellaneous tools can be used.

Function: $0403
Name: MTVersion
Returns the version number of the Miscellaneous tool set
Push: Result Space (W)
Pull: Version (W)
Errors: none
Comments: MSB is major release; LSB is minor release.

Function: $1503
Name: SysFailMgr
Displays an error message and halts the program
Push: Error Code (W); C-String Address (L)
Pull: nothing
Errors: none
Comments: A standard message is displayed if the string address parameter is 0.

Function: $2C03
Name: SysBeep
Beeps the Apple IIGS speaker
Push: nothing
Pull: nothing
Errors: none

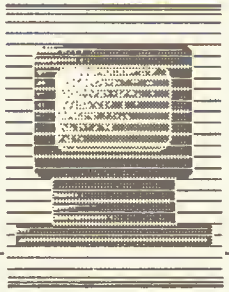
Function: $030F
Name: MenuShutDown
Shuts down the Menu Manager
Push: nothing
Pull: nothing
Errors: none

Chapter 5

A Matter of Language

As stated earlier, this book assumes that you have a strong background in programming languages, either machine language, Pascal, or C. This isn't a tutorial on programming.

Yet there's more to using a programming language and developing software than just



knowing the meaning of such terms as *ASL, printf,* or *begin.* There is a wealth of programming information to learn once you understand the basics. This information will make you a better programmer. The purpose of this chapter is to fill you in on some of the finer points of programming the IIGS, no matter which language you use.

This chapter offers programming hints and tips for the three languages covered in this book. On the following pages, you will find helpful information and suggestions for making programming and developing applications for the Apple IIGS computer much easier.

Take Life a Little Easier

Because this chapter tries to cover three very different programming environments, extra care was taken to ensure that everything was presented properly. To do that, this chapter is divided into two sections. The first section covers support files for all three languages, and the second deals with each language individually.

Support files, though they may be referenced by each language differently, are common to all three programming environments. Most amateurs avoid using support files because they don't understand them, which is a big mistake. By taking advantage of support files, you can save time and massive headaches. You should take the time to learn about support files.

The second part of this chapter concentrates on each programming language individually: The *APW* Assembler, *TML Pascal,* and C are each given a separate section. The purpose of the second half of this chapter is to help you use the language you have chosen to its full potential. After reading about Support Files, skip to the section on the language that interests you.

Of course, the adventurous reader will want to read everything: If you are only fluent in one or two languages, you may be surprised to find out what you are missing.

Support Files

To smooth the process of writing applications, the makers of *APW* and *TML Pascal* have created scores of utility and support files. These files typically contain defined routines, macros, or subroutine libraries. By taking advantage of support files, you can decrease development time and, at the same time, make your program easier to read and better looking.

In machine language, support files contain common routines and functions written as macros. For example, to avoid the redundancy of making Toolbox calls by loading the X register and performing a long jump to the subroutine at $E10000 each time a call is made, a Toolbox macro support file can be used instead. This support file already contains the defined Toolbox calls. All your source needs to do is reference the specific support file.

TML Pascal support routines, which include Apple IIGS Toolbox calls and other Pascal-oriented functions, are stored in symbolic unit files. These files end with a .USYM extension on disk. The USES keyword tells the compiler to use the unit file that corresponds to functions used in your program.

The #include directive is used to insert a source file into the compilation step when compiling a C program. Support files for C, called header files, end with a .h extension on disk. Since files used with #include can contain any instructions at all, they are far more flexible than Pascal's compile-time unit files.

The following tables illustrate how your source code could take advantage of predefined QuickDraw II functions. The following are QuickDraw II support files, each of which can be referenced by your code.

Language	**Directive**	**Support Filename**
APW Assembler	MCOPY	M16.QUICKDRAW
TML Pascal	USES	QDIntf
APW C	#include	quickdraw.h

In your source code, the above directives might take on the following syntax:

Language	**Syntax**
APW Assembler	MCOPY 2/AINCLUDE/M16.QUICKDRAW
TML Pascal	USES QDIntf;
APW C	#include <quickdraw.h>

After these statements, your source code could then use the QuickDraw II functions defined in the appropriate support file. (This will be explained in greater detail below, under each language's category.)

When programming high-level languages such as C and Pascal, these support files must be included in the compilation phase of your program to use them. Otherwise, you'll receive an *undefined function call* error message. It's best not to argue with the compiler if you want your code to run.

Macros are not required in order to make machine language Toolbox calls. The programmer can use the corresponding 65816 instructions if desired. However, using the macros defined in the *APW* Toolbox support files is accepted and a more common practice than writing out the necessary code.

The most common use for support files is to define Toolbox calls. Each tool set in the Toolbox has an associated support file. There are several other specialty and utility files, depending on your language, which can also be used to simplify writing applications.

Table 5-1 shows the support files that belong to each tool set for machine language, C, and Pascal.

Table 5-1. Tool Set Support Files

Tool Set Name	***APW* Assembler (MCOPY)**	***APW C* (#include)**	***TML Pascal* (USES)**
Tool Locator	M16.LOCATOR	locator.h	GSIntf
Memory Manager	M16.MEMORY	memory.h	GSIntf
Miscellaneous Tools	M16.MISCTOOL	misctool.h	MiscTools
QuickDraw II	M16.QUICDRAW	quickdraw.h	QDIntf
Desk Manager	M16.DESK	desk.h	GSIntf
Event Manager	M16.EVENT	event.h	GSIntf
Scheduler	M16.SCHEDULER	scheduler.h	Scheduler
Sound Manager	M16.SOUND	sound.h	Sound
DeskTop Bus	M16.ADB		
SANE	M16.SANE	sane.h	SANE
Integer Math	M16.INTMATH	intmath.h	IntMath
Text Tool Set	M16.TEXTTOOL	texttool.h	TextTools
Window Manager	M16.WINDOW	window.h	GSIntf
Menu Manager	M16.MENU	menu.h	GSIntf
Control Manager	M16.CONTROL	control.h	GSIntf
System Loader	M16.LOADER	loader.h	Loader
QuickDraw II Aux.	M16.QDAUX	qdaux.h	QDIntf
Print Manager	M16.PRINT	print.h	PrintMgr
LineEdit	M16.LINEEDIT	lineedit.h	GSIntf
Dialog Manager	M16.DIALOG	dialog.h	GSIntf
Scrap Manager	M16.SCRAP	scrap.h	GSIntf
Standard File	M16.STDFILE	stdfile.h	GSIntf
Disk Utilities			

Tool Set Name	***APW* Assembler (MCOPY)**	***APW C* (#include)**	***TML Pascal* (USES)**
Note Synthesizer	M16.NOTESYN	notesyn.h	NoteSyn
Note Sequencer	M16.NOTESEQ		
Font Manager	M16.FONT	font.h	GSIntf
List Manager	M16.LIST	list.h	ListMgr

Depending on the language you're using, there might be additional support files for working with ProDOS or a shell environment. Check your language's reference manual for more details.

At the time of this writing, some of the tool sets do not have support files, most notably those still being worked on by Apple Computer.

Although *TML Pascal*'s unit symbol files end with a .USYM extension on disk, do not include the extensions in the USES statements in your program.

In addition to the above assembler macro files, the *APW* assembler can also take advantage of *equate* files. These, like macro files, are text files that contain some of the constants and symbols listed in the Toolbox reference. For example, wAmBooli is a flag used by one of the tool sets. If your source code were using wAmBooli, as in

```
PEA    #wAmBooli
```

and if the equate file for that tool set were referenced by your source code with the COPY directive, then the assembler would replace wAmBooli with the proper value.

Table 5-2 lists the support files for equates to be used with *APW* source code. Like the macro files, they are found in the LIBRARIES/AINCLUDE subdirectory.

Table 5-2. Assembler Equate Files

Tool Set Name	**Equate File**
Tool Locator	E16.LOCATOR
Memory Manager	E16.MEMORY
Miscellaneous Tools	E16.MISCTOOL
QuickDraw II	E16.QUICDRAW
Desk Manager	E16.DESK
Event Manager	E16.EVENT
Scheduler	E16.SCHEDULER
Sound Manager	E16.SOUND
DeskTop Bus	E16.ADB
SANE	E16.SANE

Tool Set Name	Equate File
Integer Math	E16.INTMATH
Text Tool Set	E16.TEXTTOOL
Window Manager	E16.WINDOW
Menu Manager	E16.MENU
Control Manager	E16.CONTROL
System Loader	E16.LOADER
QuickDraw II Aux.	E16.QDAUX
Print Manager	E16.PRINT
LineEdit	E16.LINEEDIT
Dialog Manager	E16.DIALOG
Scrap Manager	E16.SCRAP
Standard File	E16.STDFILE
Disk Utilities	
Note Synthesizer	E16.NOTESYN
Note Sequencer	
Font Manager	E16.FONT
List Manager	E16.LIST

Note: The Disk Utilities and Note Sequencer equate files were not included in version 1.0 of the *APW* assembler.

Individual Languages

The way each language takes advantage of its support files is discussed in the following sections.

The C Language Environment

C is an elegant language, but don't let its elegance fool you. It's a nuts-and-bolts programming language. C has the detail of machine language, while retaining some of the conveniences of the high-level languages. Anyone trained in BASIC and then forced into machine language because of BASIC's crudity and slowness will enjoy C.

The road from your first *Hello World* C program to a complete application on the Apple IIGS should be smooth. Even though the *APW C* development system isn't as flashy as other programming environments, it can be used to develop large and complex applications. In fact, most of the new IIGS programs that originated on other computers are written in C, simply because the original source code can be moved to the IIGS with only minor modifications, in most cases.

Support files for *APW C* are kept in the LIBRARIES/CINCLUDE area on your *APW* program development disk. They all end with .h extensions because they are known as header files. This means that they should be included in your source code, with the #include directive, at the top (or at the head) of your program.

The following is an example of how to use a support file in a C program:

```
/* Including Header Files in C—Kinda Boring */
#include <locator.h>   /* Include the Tool Locator header file */
main( )
{
      TLStartUp( );      /* Start the Tool Locator */
      TLShutDown( );     /* Shut it down ASAP */
}
```

All the definitions for the Tool Locator functions are kept in the locator.h header file. By including this header file, the TLStartUp, TLShutDown, and other Tool Locator routines can be accessed by the C program. The same is true for any other tool set that your program uses. Include the header file for each tool set you intend to use.

```
#include <locator.h>
#include <memory.h>
#include <misctool.h>
```

The MODEL.C program, introduced in Chapter 6, has some real-life examples of support files in use.

The Pascal Environment

Pascal (not to be confused with *UCSD Pascal,* an early Apple operating system) is famous because of its structure. In fact, most educational institutions prefer to teach programming with Pascal because it forces the student to think logically and to break a problem down into smaller, easier-to-solve tasks.

Currently, the only Pascal compiler for the Apple IIGS is the one from TML Systems of Jacksonville, Florida. It's more than just a compiler. In fact, *TML Pascal* is a complete and powerful program-development system.

Support files for the *APW* version of *TML Pascal* are kept in the TOOLINTF area on your *APW* disk. The regular *TML Pascal* allows you to define where the unit files are stored. They all end with .USYM extensions because they are known unit symbol files. Rather than being included as source code as is done in C, unit symbol files are USED in *TML Pascal*. Here's an example:

```
{ Using Unit Symbol Files in Pascal }
PROGRAM Yawn;
USES QDIntF, GSIntF, MiscTools;
BEGIN
      TLStartUp;
      TLShutDown;
END.
```

The USES section of the Pascal program tells the compiler to use the unit symbol files included in the list. The corresponding functions for each tool set then become available for your program to work with.

TML doesn't intend to stop with Pascal. At this writing, they are about to release a BASIC compiler for the Apple IIGS.

The Machine Language Environment

If you're doing machine language development, you're probably using *APW*, the *Apple Programmer's Workshop*. So far, it's the most popular machine language development environment for the Apple IIGS.

To use the *APW* Assembler effectively, you'll need at least two disk drives, or one 3½-inch disk drive and a very large ramdisk of about 800K. The *APW* programs should be on one disk with your source code and any other files you need on the other. However, the best setup for any serious programming involves a hard disk with at least ten megabytes of storage. When this is the case, *APW* and all its files should be put in their own subdirectory.

The latest version of *APW* requires at least 768K of RAM on your computer, which is 512K more than the 256K that comes with the Apple IIGS .

When developing programs, it's best not to put all of your code into one, huge, cumbersome file. In fact, the best way to program is to keep your source code in small, separate modules. Not only will this help you keep track of updates (by checking the date column in a catalog listing), but it will reduce the time it takes to patch code.

The rest of the machine language examples in this book will, where applicable, use the modular concept to add pieces to the

MODEL.ASM program demonstrated in the next chapter. You can make decisions about how many modules to make, and what size to make them, on your own.

Modules are added, or chained, to one another by use of the COPY directive. For example, if the MODEL.ASM program references two other modules, DISKIO.ASM and WINDOW.ASM, the following directives should be placed at the end of the source code:

```
COPY    DISKIO.ASM
COPY    WINDOW.ASM
```

This will copy the source code from those two files to create the final program.

It is helpful to know that you're not chained to the *APW* Editor, considered by many to be a simple-minded text editor. By the time you read this, there should be several good public domain or shareware text editors on the market, any one of which could be used to edit *APW* source files.

Using *APW* is similar to using MS-DOS or UNIX. However, programs created under *APW* are not directly executable by the Finder or Launcher. You must change their filetype from an EXE ($B5) to a S16 ($B3) file type. This is done with the FILETYPE command at the *APW* system prompt. For example,

```
FILETYPE MODELA S16
```

changes the file type of the MODELA program from EXE to S16, allowing the program to be run directly from the Finder or Launcher.

Other than that, *APW* is straightforward and easy to use, considering that you're writing machine language. However, there is one more detail about the *APW:* Machine language programmers should pay special attention to the way the *APW* assembler uses macros.

Macros. *Macro* is a an abbreviation of *macroinstruction.* A macro is used to represent a number of other statements, like an abbreviation. Some complex macros can even make decisions and perform evaluations. Yet, you only write the macro instruction once. Then, from that point on, you use only the name of the macro to reference it.

You probably won't find any machine language examples in any books that don't use macros (other than *Mastering the Apple IIGS Toolbox,* where macros were not used in order to better explain certain concepts). Because of this fact, all the machine language source code in this book uses macros. You'll find that macros make programs easier to read and easier to write. For this reason, the rest of this chapter is devoted to *APW* machine language macros and how to use them.

Macro etiquette. Macros are most commonly used to make Toolbox calls. With the *APW* assembler, the convention for Toolbox macros is to start them with the underscore character as in the following example that invokes the MoveTo Toolbox call:

```
_MoveTo
```

Case is unimportant as far as macros are concerned, unless you've specifically told the *APW* assembler to pay attention to case by using the CASE ON directive. Each of the following lines will invoke the MoveTo macro (assuming you have not used the CASE ON directive):

```
_moveto
_MOVETO
_MoVeTo
```

All Toolbox calls have a macro, as defined in the support files in the LIBRARIES/AINCLUDE subdirectory. And all Toolbox macros carry the same name as their Toolbox function, with each preceded by an underscore.

Aside from the Toolbox calls, several other *APW* assembler macro types are popular. The ones most often seen are these:

Macro	**Action**
PushLong	Push a long-word value onto the stack
PushWord	Push a word value onto the stack
PullLong	Pull a long-word value from the stack
PullWord	Pull a word value from the stack
Str	Create a Pascal string

There are some distinct advantages to using the PushLong and PushWord macros over the PEA instructions. The most common is the error that occurs when a memory location rather than a value is pushed to the stack (PEA $1234 instead of PEA #$1234). If you use the PushLong and PushWord macros, there will be no question about which type is being pushed.

Also, the Str macro eliminates some of the tedious labeling that occurs when defining a Pascal string. (Pascal strings start with a count byte to tell the program how many characters to expect.) Because Pascal strings are used frequently in the Toolbox, the Str macro is very handy.

Macros at work. When the assembler sees your macro, it expands the macro into the code it stands for. This is one of the most confusing aspects of using macros.

Macros make the source code easier to read and easier to debug. They help the programmer avoid redundancy by eliminating the need to type the same code repeatedly. A beginning machine language programmer might assume that using macros tightens up code. That's only half true: Macros make your source code tighter, but your object code will be just as long as if you didn't use macros.

When your source code is assembled into object code, the macros you use are expanded out into their raw form. So for each _MMStartUp the assembler sees, it replaces it with the appropriate code:

```
ldx   #$0202
jsl   #E10000
```

Macros can be simple (as above) or complex. For example, a macro can look rather innocent in the middle of your source code:

```
PushLong   #1234
```

The PushLong macro is much more complex than _MMStartUp. PushLong will be translated by the assembler into the codes defined in the macro. Because there can be a number of arguments for PushLong (a value, a memory location, or a zero-page location plus an offset, the stack plus an offset, and so on), the PushLong macro must make a few decisions.

The actual definition for the PushLong macro is quite complex:

```
        MACRO
&lab    pushlong &addr,&offset
&lab    ANOP
        LCLC        &C
        LCLC        &REST
&C      AMID        &addr,1,1
        AIF         &C=#,.immediate
        AIF         &C=[,.zeropage
        AIF         C:&offset=0,.nooffset
        AIF         &offset=s,.stack
```

```
          pushword   &addr+2,&offset
          pushword   &addr,&offset
          MEXIT
.nooffset
          pushword   &addr+2
          pushword   &addr
          MEXIT
.immediate
&REST     AMID       &addr,2,L:&addr-1
          dc         I1‘$F4’,I2‘(&REST)|-16’
          dc         I1‘$F4’,I2‘&REST’
          MEXIT
.stack
          pushword   &addr+2,s
          pushword   &addr+2,s
          MEXIT
.zeropage
          ldy        #&offset+2
          pushword   &addr,y
          ldy        #&offset
          pushword   &addr,y
          MEND
```

Inside PushLong's definition are conditional branches and evaluations to determine exactly what type of long-word value is being pushed on the stack. The assembler, when it replaces the macro PushLong with the above instructions, will make certain evaluations and then use only those instructions to push the proper long value onto the stack.

For example, if

```
PushLong   #1234
```

is specified in your source, the assembler will use the following instructions from the PushLong macro to push #1234 onto the stack:

```
+          ANOP
+          LCLC     &C
+          LCLC     &REST
+&C        AMID     #1234,1,1
+.immediate
+&REST     AMID     #1234,2,L:&addr-1
+          dc       I1‘$F4’,I2‘(1234)|-16’
+          dc       I1‘$F4’,I2‘1234’
```

That seems like a very complex procedure to go through just to push a long word on the stack, yet some complex decision making is occurring. For PushLong to be a versatile macro, capable of pushing a variety of values onto the stack, it has to be complex. Fortunately, the logic and debugging of the PushLong macro has been taken care of for you. You need only specify it in your source and let the assembler do the rest.

To see how a macro expands, the TRACE ON directive can be listed at the top of your assembler source. By adding the LIST ON directive, you'll be able to see your source code as it's assembled and, with TRACE ON set, see the macros expanded as well.

Using macros in your source code. With the *APW* Assembler, macros exist in an external file and are referenced in your source code by the MCOPY directive:

```
MCOPY    [pathname]
```

The pathname is the name of a path or file that contains all your program's macro definitions. For example:

```
MCOPY    MYMACROS
```

The above instruction directs the assembler to look for any macro references in the file MYMACROS. Make sure you have this statement at the top of your source code. The macros used by your source cannot be accessed until the assembler has encountered the MCOPY command.

So, if your source code makes extensive use of QuickDraw II Toolbox calls, and you want to use the QuickDraw II macros supplied with *APW*, you could place the following at the top of your source code:

```
MCOPY    /APW/LIBRARIES/AINCLUDE/M16.QUICKDRAW
```

Because *APW* takes advantage of ProDOS 16 prefix numbers, you can substitute the number 2 for /APW/LIBRARIES above. (No matter what the configuration of your drive, using 2 will work. See the section in the *APW* manual about the LOGIN file for more information.)

```
MCOPY    2/AINCLUDE/M16.QUICKDRAW
```

After using this instruction, any QuickDraw II macros referenced in your source code will be replaced by the definitions in the M16.QUICKDRAW support file.

This sounds like a powerful feature when you're reading the *APW* Assembler manual and might cause you to think you could just MCOPY *all* the predefined macros in the AINCLUDE subdirectory into your source code. While that sounds logical, and it would make things easier, it's just not the case.

The MCOPY command only allows four macro files to be in use at one time. The manual seems to suggest that you can juggle these four macro files using the MLOAD and MDROP directives throughout your code. However, this is a fallacy. Rather than toss about MCOPY, MDROP, and MLOAD directives, it's much faster and easier to create a custom macro file for your source code files. This is done with the MACGEN program from the *APW* shell:

```
MACGEN [source.code] [macro.file] [macro.libraries . . .]
```

MACGEN creates a custom macro file for your source code. It's one of *APW*'s better utilities.

First, MACGEN reads in your entire source file. Then, it scans a specified list of macro support files, pulls out only the macros referenced by your source code, and finally creates a custom macro file containing only the macros referred to by your source.

For example, consider the program MODEL.ASM in the next chapter. MODEL makes extensive use of macros. Most of those macros are defined by the M16 files in the AINCLUDE prefix. To build a custom macro file containing only the macros referenced by MODEL.ASM, the following MACGEN command was typed at the *APW* system prompt:

```
MACGEN model.asm model.macros 2/ainclude/m16.=
```

This reads: From the source code model.asm, generate a macro file named model.macros using all the files that start with m16. in the subdirectory AINCLUDE. (The equal sign is a wildcard specifying all files starting with M16.)

MACGEN reads in the source code and then reads through all the files M16.= for any matching macros. It then places the macros it finds into the file MODEL.MACROS. To take advantage of them, the following is placed at the start of MODEL.ASM:

```
MCOPY    MODEL.MACROS
```

If your program has more than one module, you should use the MACGEN command on the main module. As long as the main module has COPY or APPEND directives, the other related source file modules will also be scanned for macro references.

Building a custom macro file with the aid of MACGEN is the best way to provide the macros your program needs. If you update your source listing with new macro calls, you can run MACGEN a second time to create a new custom macro file. Also, any unique macros you create can be typed into the macro file using the *APW* editor.

Summary

Macro files, header files, unit symbol files, nearly all the information covered in this chapter can be found elsewhere. However, many people who consider themselves old hands at programming have never taken advantage of support files. With the Apple IIGS Toolbox at your disposal, using support files and paying attention to the tips offered in this chapter can make you a better programmer.

Chapter 6

The DeskTop

Applications for the Apple IIGS fall into two categories: DeskTop and non-DeskTop. This chapter introduces some new and exciting things happening in the DeskTop world of programming. It begins with a description of a DeskTop program and provides a sample program, in three languages, that you can run on your computer.

The DeskTop

A DeskTop program is one that takes advantage of the 16-bit processing power of the Apple IIGS and uses its built-in tools to manipulate pull-down menus, windows, dialog boxes, icons, the mouse, and so on. This interface has proven to be highly intuitive to the user and is popular on a variety computers. DeskTop programs written for the Apple IIGS will not run on the Apple IIe or IIc.

A non-DeskTop program is one written for the eight-bit personality of the Apple IIGS. This half of the computer, also called the Mega II, emulates an Apple IIe with 128K of RAM and a 65C02 processor. Programs in that environment rarely use the powerful tools that reside in the Apple IIGS Toolbox ROM. They are required to provide their own memory-management schemes and custom interfaces. This entails a lot of work for the programmer. However, these programs can run on the Apple IIGS as well as on the Apple IIe and IIc.

Having a "canned interface" inside the computer provides many advantages. Users feel at home with DeskTop programs because the interface is consistent from one program to the next. Programmers can concentrate on the tasks of their software and are spared the details of interacting with the user. Since most of the code for the interface resides in ROM, programs require only a few calls to drive the entire DeskTop.

The DeskTop interface, remarkably similar to that found in Apple's Macintosh computer, is the most exciting aspect of the Apple IIGS.

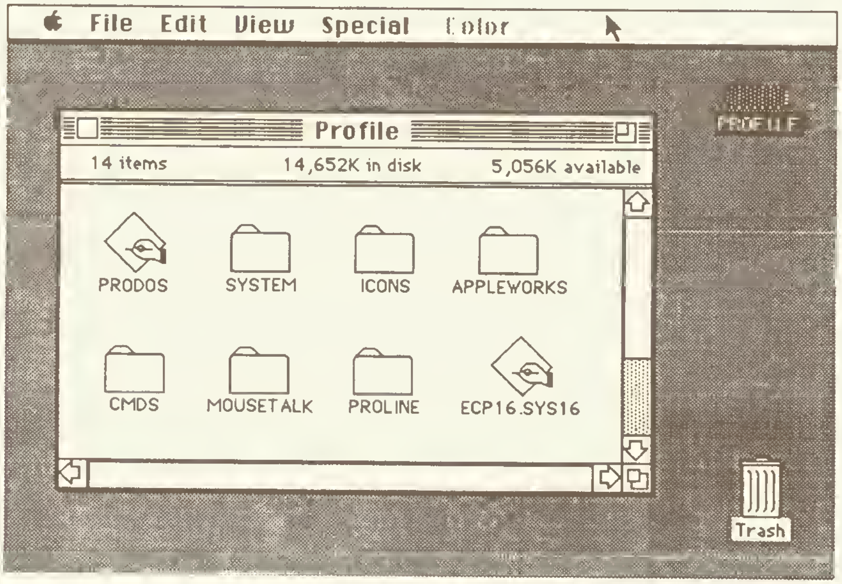
Managers

Here's a quick description of the Apple IIGS DeskTop and how the various managers built into the IIGS are responsible for maintaining it.

When a DeskTop program is first launched, a blank background pattern is displayed across the entire Apple IIGS super-hires graphics screen. Traditionally, the background pattern is a solid shade of light blue, though the programmer can choose any color supported by computer.

Inevitably, the DeskTop will have a menu bar at the top of the screen which contains the titles of one or more pull-down menus. These menus contain all of the program's commands and functions available to the user.

Figure 6-1. Figure of DeskTop with Menus, Dialog Boxes, and Windows


File
Edit
View
Special
Color
PROFILE
Profile
14 items
14,652K in disk
5,056K available
PRODOS
SYSTEM
ICONS
APPLEWORKS
CMDS
MOUSETALK
PROLINE
ECP16.SYS16
Trash

It is the responsibility of the Event Manager to track the location of the mouse and update the mouse pointer on the screen. The mouse pointer, an arrow shape, marks the position on the screen where the mouse is located on the DeskTop. Moving the physical mouse device will cause the mouse pointer to move accordingly on the screen. This function is completely transparent to the DeskTop application because it relies on the interrupt feature of the Apple IIGS microprocessor.

The mouse is used to select items on the DeskTop. For example, the user moves the mouse pointer over a title on the menu bar and presses the mouse button. This causes a pull-down menu to be displayed, showing a list of available selections. By holding down the mouse button and moving the pointer (an action called *dragging*) the user chooses a menu item from the menu. A selection is made when the mouse button is released.

The programmer organizes what is to be placed into the menus and passes that information along to the Menu Manager. The job of drawing pull-down menus and interacting with the user while a selection is made is handled completely by the Menu Manager. To do this, of course, it relies on other tool sets, especially QuickDraw II.

Some menu items are selected with the keyboard instead of the mouse. This is done by pressing the Open Apple key in conjunction with another key that corresponds to a menu item. The user determines if a menu item has a keyboard equivalent by examining the list of items in a pull-down menu. Menu items with keyboard equivalents have apple symbols, followed by the command character, after their name in the menu.

For the programmer, all of the work involved in getting the user's selections via the mouse or keyboard equivalents is handled by the routines in the Toolbox. It's the job of the Window Manager's TaskMaster function to manage these details.

After a menu item is selected, any number of events might occur. As an example, a dialog box could be displayed asking the user to supply input for the application. Appropriately named, dialog boxes let the user communicate with the DeskTop program by filling in blank entries with text, turning switches on or off, pressing buttons, or by using other controls.

Using software, the programmer builds the dialog box to the required specifications. Buttons and other controls can be installed on the box. The functions in the Dialog Manager and Control Manager allow the user to manipulate the controls and report to the application which buttons have been pressed.

The function of a typical DeskTop program is as simple as making a selection from a vending machine. The user makes selections from the menu bar and interacts with a few dialog boxes, and the computer performs its assigned task.

The Apple IIGS has more managers than a small baseball league. The programmer is well assisted in driving the DeskTop.

Parts of a DeskTop Program

At the software level, DeskTop applications consist of three main parts:

Startup	Before a program can begin to interact with the user, it must complete the startup phase. This involves starting a host of tool sets, allocating memory, and setting up the DeskTop environment with pull-down menus and so forth.

Event handling	Once everything is initialized, a DeskTop program basically sits idle, waiting for the user to make selections from the pull-down menus. When a menu item is selected, a corresponding function for that item is dispatched and carried out.
Shutdown	Eventually, the user will be finished with the program and will want to quit. As part of the shutdown process, the application will take care of unfinished business, such as saving changes to disk. It shuts down the tool sets it started up, deallocates reserved memory, and exits to the operating system.

These three steps provide the basic framework of practically every DeskTop program written. The nice thing about this is that once you've created the *overhead code* (the basic code that performs these three functions), it can be used over and over again for new programs.

The Tower of Babel

The following sample program—shown here in *APW* machine language source code, *APW C*, and *TML Pascal*—demonstrates how a typical DeskTop program starts up, handles events, and shuts down. It doesn't do anything spectacular. But it sets the stage for some very exciting programming ventures using the powerful abilities of the Apple IIGS Toolbox.

Referring to these programs as models, the next few chapters will describe the important details in creating DeskTop programs. Study closely the program listing written in the language you're most interested in.

Program 6-1. MODEL.ASM

```
*-------------------------------------------------------*
*                     MODEL.ASM                         *
*  Sample Desktop Application in APW Assembler (1.0)    *
*-------------------------------------------------------*

; To create the ModelMacs macro file, use this APW shell command:
; # macgen model.asm modelmacs 2/ainclude/m=

        ABSADDR ON
        KEEP    ModelA
        MCOPY   ModelMacs

*-------------------------------*
*       Global Equates          *
*-------------------------------*
```

```
Toolbox gequ    $e10000         ;Primary tool dispatcher
TRUE    gequ    $8000           ;True value
FALSE   gequ    $0000           ;False value
Page    gequ    $100            ;The size of a page (256 bytes)

ModelA  START
        phk                     ;Make the data bank...
        plb                     ;...the current code bank
        brl     Main            ;branch over functions to Main

*------------------------------*
*    Handle Toolbox Errors     *
*------------------------------*

ErrChk  bcs     Die             ;Carry set if error

        rts                     ;Else, return

Die     pha                     ;Toolbox returns error in A
        pushlong #0             ;Use standard system death message
        _SysFailMgr             ;Get ready to slide apples back and forth

*------------------------------*
*  Manage Direct Page Buffers  *
*------------------------------*
; Returns address of next free Direct Page. (Modifies Y register)
; The GetDPs entry point requires byte count in A register.

GetDP   lda     #Page           ;Ask for one 256 byte DP block
GetDPs  clc                     ;Alternate entry: A = Number of bytes
        ldy     DPBase          ;Get base value (we return this)
        add     ,DPBase,DPBase  ;Add A to our last DP buffer address
        tya                     ;Return entry value
        rts

*------------------------------*
*       Start Up Tools         *
*------------------------------*

DPSpace equ     $000600         ;Memory needed for direct pages
HndlRef equ     $00             ;direct page handle deref pointer

StartUpTools    anop
        _TLStartUp              ;----Start the Tool Locator

        pha                     ;Result Space for User ID
        _MMStartUp              ;----Start the Memory Manager
        jsr     ErrChk          ;Check for errors
        pullword UserID         ;Get our User ID and save it
        ora     #%100000000     ;munge an auxiliary ID...
        sta     MemID           ;used for Memory Manager handle usage

        _MTStartUp              ;----Start the Misc Toolset

; Get direct page space for other tools

        pha                     ;Long result space...
        pha                     ;...for returned handle
        pushlong #DPSpace       ;Long value: size of memory block
        pushword MemID          ;Use the special ID for handle allocation
pushword #$c005          ;Fixed, Page-aligned, Locked, Unpurgable
pushlong #$000000        ;Where our block resides ($00/0000)
_NewHandle
jsr      ErrChk          ;Check for errors
pullong HndlRef          ;Get handle of new block
lda      [HndlRef]       ;Get address of the storage area
sta      DPBase          ;Save it for GetDP utilities

lda      #3*Page         ;QuickDraw requires 3 direct pages
jsr      GetDPs          ;Get address for them...
pha                      ;...and push it
pushword #$0080          ;Screen Mode (use $0000 for 320 mode)
pushword #$00a0          ;Pixel Map Size (use $0050 for 320 mode)
pushword UserID          ;Push our program's ID
_QDStartUp               ;----Start QuickDraw II
jsr      ErrChk

jsr      GetDP           ;Requires a Direct Page
pha                      ;Push the DP address
pushword #20             ;Event queue size
pushword #0              ;Min X clamp
pushword #640            ;Max X clamp (640 mode)
pushword #0              ;Min Y clamp
pushword #200            ;Max y clamp = 200 (bottom of screen)
pushword UserID          ;Push program's User ID
_EMStartUp               ;----Start the Event Manager
jsr      ErrChk
pushword #0              ;Standard background color
_SetBackColor

pushword #3              ;Standard foreground color
_SetForeColor

pushword #260            ;X position of message
pushword #85             ;Y position of message
_MoveTo                  ;Move pen to X,Y

pushlong #Moment         ;Point to a message string
_DrawCString             ;Print "One moment..."

pushlong #Toolist        ;Point to a list of tools
_LoadTools               ;Read tools from disk into RAM
jsr      ErrChk

pushword UserID          ;Requires User ID
_WindStartUp             ;----Start the Window Manager
jsr      ErrChk

pushword UserID          ;Requires User ID...
jsr      GetDP           ;...and a Direct Page
pha
_CtlStartUp              ;Start the Control Manager
jsr      ErrChk

pushword UserID          ;Requires User ID...
jsr      GetDP           ;...and a Direct Page
pha
_MenuStartUp             ;----Start the Menu Manager
jsr      ErrChk

_DeskStartUp             ;----Start the Desk Manager
rts
```

```
*------------------------------*
*   Prepare Desktop and Menus  *
*------------------------------*

PrepDeskTop     anop
        pushlong #0             ;Draw entire desktop using default values
        _RefreshDesktop
        _InitCursor

NxtMenu pha                     ;Result Space (Long) for...
        pha                     ;...the menu's handle
        lda     MenuTbl         ;Get menu count
        asl     A               ;x 2
        tax                     ;Make it an index into word values
        lda     MenuTbl,x       ;Get address of menu structure
        phb                     ;Push program bank twice
        phb                     ;(PHB pushes only a byte)
        pha                     ;Push address of menu structure
        _NewMenu                ;the menu handle is now on the stack
        pushword #0             ;Insert menu at left, shifting right
        _InsertMenu

        dec     MenuTbl         ;More menus to install?
        bne     NxtMenu         ;Yes

        pushword #1             ;Put Desk Accessories's in Apple Menu
        _FixAppleMenu

        pha                     ;Result space
        _FixMenuBar             ;Calculate menu bar's height
        pla                     ;Discard height for now

        _DrawMenuBar            ;Display the menu bar
        rts

*------------------------------*
* Apple Menu: About            *
*------------------------------*

About   rts                     ;Does nothing (for now)

*------------------------------*
* File Menu: Quit              *
*------------------------------*

Quit    dec     QFlag           ;User wants to quit (QFlag = $ffff)
        rts

*------------------------------*
*      Do Menu Selection       *
*------------------------------*

DoMenu  lda     TaskData        ;Get TaskData Item ID number
        and     #$00ff          ;Discard upper 8-bits
        asl     A               ;Double the value
        tax
        jsr     (MTable,x)      ;Dispatch the proper menu item handler

        pushword #FALSE         ;We need to unHilite the menu title now
        pushword TaskData+2     ;Get TaskData Menu number
```

```
         _HiliteMenu              ;Unhilite menu title
         rts

*------------------------------*
*      Shutdown Toolsets       *
*------------------------------*

ShutDownTools   anop
         lda     Toolist          ;Get # of toolsets started up
         asl     A                ;x2
         asl     A                ;x2 (to create index over longwords)
         tax
         lda     Toolist-2,x      ;Get toolset ID from list
         cmp     #$0002           ;Memory Manager?
         bne     Shut1            ;No, so shut this down right now
         pushword MemID           ;Dispose all handles allocated
         _DisposeAll
         pushword UserID          ;Shut down this program's memory
         lda     #$0002           ;MMShutDown
Shut1    ora     #$0300           ;Make it a shutdown call
         tax                      ;Set X to call number
         jsl     Toolbox          ;Shut it down!
         dec     Toolist          ;Shutdown another toolset?
         bne     ShutDownTools    ;Yes

         rts

*------------------------------*
*          Main                *
*------------------------------*

Main     jsr     StartUpTools     ;Start toolsets
         jsr     PrepDeskTop      ;Prepare desktop and menus

Scan     pha                      ;Result Space
         pushword #$ffff          ;Event Mask
         pushlong #EventRec       ;Point to Event Record
         _TaskMaster
         pla                      ;Get HandleEvent flag
         beq     Scan             ;If nothing, continue looping

         cmp     #$11             ;A menu event? ($11=wInMenuBar)
         bne     Scan             ;Nope, just keep scanning

         jsr     DoMenu           ;Do menu item dispatch
         bit     QFlag            ;Time to quit?
         bpl     Scan             ;No, keep scanning for events

         jsr     ShutDownTools    ;Shut down all tools started

         _QUIT   Qparms           ;Exit this program through ProDOS 16

*------------------------------*
*      Variable Storage        *
*------------------------------*

UserID   ds      2                ;Our User ID
MemID    ds      2                ;Memory User ID (made from User ID)
DPBase   ds      2                ;Used by DP buffer manager
QFlag    dc      i'FALSE'         ;Boolean: Quit flag (starts out as false)
```

```
*------------------------------*
*  StartUp/Shutdown Tool List  *
*------------------------------*

Toolist dc      i'(ToolstE-Toolist-1)/4'        ;Tool count
        dc      i'1,0'          ; Tool Locator
        dc      i'2,0'          ; Memory Manager
        dc      i'3,0'          ; Misc Tools
        dc      i'4,0'          ; QuickDraw II
        dc      i'6,0'          ; Event Manager
        dc      i'14,0'         ; Window Manager
        dc      i'16,0'         ; Control Manager
        dc      i'15,0'         ; Menu Manager
        dc      i'5,0'          ; Desk Manager
ToolstE         anop
*------------------------------*
*   Pull Down Menu Structures  *
*------------------------------*

MenuTbl dc      i'(MenTblE-MenuTbl-1)/2'        ;Menu count
        dc      i'Menu1'        ;Apple
        dc      i'Menu2'        ;File
        dc      i'Menu3'        ;Edit
MenTblE anop

Menu1   dc      c'>>@\XN1',i1'0'                ;Apple
        dc      c'--About This Program...\N256',i1'0'
        dc      c'---\D',i1'0'
        dc      c'>'

Menu2   dc      c'>>  File  \N2',i1'0'          ;File
        dc      c'--Quit\N257*Qq',i1'0'
        dc      c'>'

Menu3   dc      c'>>  Edit  \N3D',i1'0'         ;Edit
        dc      c'--Undo\N250V*Zz',i1'0'
        dc      c'--Cut\N251*Xx',i1'0'
        dc      c'--Copy\N252*Cc',i1'0'
        dc      c'--Paste\N253V*Vv',i1'0'
        dc      c'--Clear\N254',i1'0'
        dc      c'>'

*------------------------------*
* Menu Item Dispatch Addresses *
*------------------------------*

MTable  dc      i'About'        ;256/About      (Apple Menu)
        dc      i'Quit'         ;257/Quit       (File Menu)

*------------------------------*
*       The Event Record       *
*------------------------------*

EventRec        anop                    ;Event Record used by TaskMaster
EWhat           ds      2               ;What
EMsg            ds      4               ;Message
EWhen           ds      4               ;When
EWhere          ds      4               ;Where
EMods           ds      2               ;Modifiers
TaskData        ds      4               ;Task Data
TaskMask        dc      i4'$1fff'       ;Task Mask
```

```
*------------------------------*
*      Miscellaneous Data      *
*------------------------------*

Moment  dc      c'One Moment...',i1'0'

QParms  dc      i4'0'           ;ProDOS 16 Quit Code parameters
        dc      i'$0000'

        END
```

The sample program written in *APW* machine language will create a four-block object file on disk. Of that, one block (512 bytes) of header information is used for the System Loader. The last three blocks contain the actual machine language program.

Program 6-2. MODEL.C

```
/*--------------------------------------------*
 *                  MODEL.C                    *
 * Sample Desktop Application in APW C (1.0)   *
 *--------------------------------------------*/

#include <types.h>
#include <stdio.h>
#include <locator.h>
#include <memory.h>
#include <misctool.h>
#include <quickdraw.h>
#include <event.h>
#include <window.h>
#include <menu.h>
#include <control.h>
#include <desk.h>

/*------------------------------*
 *       Global Variables       *
 *------------------------------*/

WmTaskRec   EventRec;           /* Event Record Structure */

Word        Event,              /* Event code */
            UserID,             /* Our User ID */
            MemID,              /* Memory Management ID */
            QFlag;              /* Boolean: Quit flag */

Word        Toolist[] = {
                3,              /* Tool count */
                14, 0,          /* Window Manager */
                15, 0,          /* Menu Manager */
                16, 0           /* Control Manager */
};

char        *DPBase;            /* Direct Page base pointer */

/*------------------------------*
 *    Handle Toolbox Errors     *
 *------------------------------*/
```

```
ErrChk()                        /* Check for error, die if so */
{
    if (_toolErr) SysFailMgr(_toolErr, nil);
}

/*-------------------------------*
 *  Manage Direct Page Buffers   *
 *-------------------------------*/

char    *GetDP(bytes)
Word    bytes;
{
    char *OldDP = DPBase;
    DPBase += bytes;            /* Update base level pointer */
    return (OldDP);             /* Return old DPBase pointer */
}

/*-------------------------------*
 *       Start Up Tools          *
 *-------------------------------*/

StartUpTools()
{
    Word    GetDP();        /* Force words from GetDP */

    TLStartUp();
    UserID = MMStartUp();                                           ErrChk();
    MemID = UserID | 256;
    MTStartUp();
    DPBase = *(NewHandle(0x600L, MemID, 0xc005, nil));              ErrChk();
    QDStartUp(GetDP(0x300), 0x80, 0xa0, UserID);                    ErrChk();
    EMStartUp(GetDP(0x100), 0x14, 0, 0x280, 0, 0xc8, UserID);   ErrChk();

    SetBackColor(0);                /* Show Intro Screen */
    SetForeColor(3);
    MoveTo(0x104, 0x55);
    DrawCString("One Moment...");

    LoadTools(Toolist); ErrChk();   /* Load & Startup tools */

    WindStartUp(UserID);                ErrChk();
    CtlStartUp(UserID, GetDP(0x100));   ErrChk();
    MenuStartUp(UserID, GetDP(0x100));  ErrChk();
    DeskStartUp();
}

/*-------------------------------*
 *  Prepare Desktop and Menus    *
 *-------------------------------*/

PrepDeskTop()
{

    static char *AppleMenu[] = {
        ">>@\\XN1",
        "--About This Program...\\N256",
        "---\\D",
        ">"
    };
    static char *FileMenu[] = {
        ">>  File  \\N2",
        "--Quit\\N257*Qq",
        ">"
    };

    static char *EditMenu[] = {
        ">>  Edit  \\N3D",
        "--Undo\\N250V*Zz",
        "--Cut\\N251*Xx",
        "--Copy\\N252*Cc",
        "--Paste\\N253V*Vv",
        "--Clear\\N254",
        ">"
    };

    RefreshDesktop(nil);                        /* Display Desktop */
    InitCursor();                               /* Show mouse cursor */

    InsertMenu(NewMenu(EditMenu[0]), 0);        /* Install menus */
    InsertMenu(NewMenu(FileMenu[0]), 0);
    InsertMenu(NewMenu(AppleMenu[0]), 0);

    FixAppleMenu(1);                            /* Display menu bar */
    FixMenuBar();
    DrawMenuBar();
}

/*-------------------------------*
 *  Apple Menu: About            *
 *-------------------------------*/

About()
{
        /* Does nothing (for now) */
}

/*-------------------------------*
 *      Do Menu Selection        *
 *-------------------------------*/

DoMenu()
{
    switch(EventRec.wmTaskData) {
    case 256:   About;                 /* Apple Menu: About */
                break;
    case 257:   QFlag = TRUE;          /* File Menu: Quit */
                break;
    }

    HiliteMenu(FALSE, EventRec.wmTaskData>>16);
}

/*-------------------------------*
 *      Shutdown Toolsets        *
 *-------------------------------*/
```

```
ShutDownTools()
{
    DeskShutDown();
    MenuShutDown();
    CtlShutDown();
    WindShutDown();
    EMShutDown();
    QDShutDown();
    MTShutDown();
    DisposeAll(MemID);
    MMShutDown(UserID);
    TLShutDown();
}
/*------------------------------*
 *           Main               *
 *------------------------------*/

main()
{
    StartUpTools();             /* Start toolsets */
    PrepDeskTop();              /* Prepare desktop and menus */

    QFlag = FALSE;
    EventRec.wmTaskMask = 0x00001fff;

    while (!QFlag) {            /* Wait for a menu event */
        do {
            Event = TaskMaster(0xffff, &EventRec);
        } while (!Event);
        if (Event == wInMenuBar) DoMenu();
    }

    ShutDownTools();            /* Shutdown all tools started */
    exit(0);
}
```

The sample program written in *APW C* compiles into a 16-block object file. However, a compiled C program containing no instructions at all produces a 12-block file. This means that, like the assembly program, about 4 blocks contain the actual code, while the other 12 consist mostly of overhead from the standard C library and System Loader.

Program 6-3. MODEL.PAS

```
{ *-------------------------------------------------*
  *                   MODEL.PAS                     *
  * Sample Desktop Application in TML Pascal (v1.01) *
  *-------------------------------------------------* }

PROGRAM ModelP;

USES    QDIntF,
        GSIntF,
        MiscTools;
```

```
{ *-----------------------------*
  *      Global Variables       *
  *-----------------------------* }

VAR EventRec:   EventRecord;    { Taskmaster Structure }
    Event:      Integer;        { Event code }
    UserID:     Integer;        { Our User ID }
    MemID:      Integer;        { Memory allocation ID }
    DPBase:     Integer;        { Direct Page base pointer }
    QFlag:      Boolean;        { Boolean: Quit flag }

    AppleMenu:  String;         { Pull down menu strings }
    FileMenu:   String;
    EditMenu:   String;

{ *-----------------------------*
  *    Handle Toolbox Errors    *
  *-----------------------------* }

PROCEDURE ErrChk;               { Check for error, die if so }
BEGIN
    IF IsToolError THEN
        SysFailMgr(ToolErrorNum, 'Tool error -> $');
END;

{ *-----------------------------*
  *  Manage Direct Page Buffers *
  *-----------------------------* }

FUNCTION GetDP(bytes: Integer): Integer;
BEGIN
    GetDP := DPBase;            { Return current DPBase }
    DPBase := DPBase + bytes;   { Update base level pointer }
END;

{ *-----------------------------*
  *       Start Up Tools        *
  *-----------------------------* }

PROCEDURE StartUpTools;
VAR

    Toolist:    ToolTable;      { Disk-based tool list }
    Height:     Integer;        { Menu bar heigth (unused) }

BEGIN
    Toolist.NumTools := 3;                  { Tool count }
    Toolist.Tools[1].TSNum := 14;           { Window Manager }
    Toolist.Tools[1].MinVersion := 0;
    Toolist.Tools[2].TSNum := 15;           { Menu Menager }
    Toolist.Tools[2].MinVersion := 0;
    Toolist.Tools[3].TSNum := 16;           { Control Manager }
    Toolist.Tools[3].MinVersion := 0;

    TLStartUp;
    UserID := MMStartUp;                                          ErrChk;
    MemID := UserID + 256;
    MTStartUp;
    DPBase := LoWord(NewHandle($600, MemID, $c005, Ptr(0))^);    ErrChk;
    QDStartUp(GetDP($300), $80, $a0, UserID);                     ErrChk;
    EMStartUp(GetDP($100), $14, 0, $280, $0, $c8, UserID);        ErrChk;
    SetBackColor(0);                { Show Intro Screen }
    SetForeColor(3);
    MoveTo($104, $55);
    DrawString('One Moment...');

    LoadTools(Toolist); ErrChk;     { Load & Startup tools }

    WindStartUp(UserID);                ErrChk;
    CtlStartUp(UserID, GetDP($100));    ErrChk;
    MenuStartUp(UserID, GetDP($100));   ErrChk;
    DeskStartup;
END;

{ *------------------------------*
  *  Prepare Desktop and Menus   *
  *------------------------------* }

PROCEDURE PrepDeskTop;
VAR
    Height:     Integer;

BEGIN
    AppleMenu := CONCAT('>>@\XN1\0',
                        '--About This Program...\N256\0',
                        '---\D\0',
                        '>');

    FileMenu := CONCAT('>>  File  \N2\0',
                       '--Quit\N257*Qq\0',
                       '>');

    EditMenu := CONCAT('>>  Edit  \N3D\0',
                       '--Undo\N250V*VZz\0',
                       '--Cut\N251*Xx\0',
                       '--Copy\N252*Cc\0',
                       '--Paste\N253V*Vv\0',
                       '--Clear\N254\0',
                       '>');

    Refresh(Nil);                           { Display Desktop }
    InitCursor;                             { Show mouse cursor }

    InsertMenu(NewMenu(@EditMenu[1]), 0);   { Install menus }
    InsertMenu(NewMenu(@FileMenu[1]), 0);
    InsertMenu(NewMenu(@AppleMenu[1]), 0);

    FixAppleMenu(1);                        { Display menu bar }
    Height := FixMenuBar;
    DrawMenuBar;
END;

{ *------------------------------*
  *  Apple Menu: About           *
  *------------------------------* }

PROCEDURE About;
BEGIN
        { Does nothing (for now) }
END;
```

```
{ *------------------------------*
  *       Do Menu Selection       *
  *------------------------------* }

PROCEDURE DoMenu;
BEGIN
    CASE LoWord(EventRec.TaskData) OF
        256:    About;                  { Apple Menu: About }
        257:    QFlag := TRUE;          { File Menu: Quit }
    END;

    HiliteMenu(FALSE, HiWord(EventRec.TaskData));
END;

{ *------------------------------*
  *      Shutdown Toolsets        *
  *------------------------------* }

PROCEDURE ShutDownTools;
BEGIN
    DeskShutDown;
    MenuShutDown;
    CtlShutDown;
    WindShutDown;
    EMShutDown;
    QDShutDown;
    MTShutDown;
    DisposeAll(MemID);
    MMShutDown(UserID);
    TLShutDown;
END;

{ *------------------------------*
  *            Main               *
  *------------------------------* }

BEGIN
    StartUpTools;               { Start toolsets }
    PrepDeskTop;                { Prepare desktop and menus }

    QFlag := FALSE;
    EventRec.TaskMask := $00001fff;

    REPEAT                      { Wait for a menu event }
        REPEAT
            Event := TaskMaster($ffff, EventRec);
        UNTIL Event <> 0;
        IF Event = wInMenuBar THEN DoMenu;
    UNTIL QFlag;

    ShutDownTools               { Shutdown all tools started }
END.
```

Surprisingly, the *TML Pascal* example compiles into an eight-block runtime file, half the size of the C program.

These sample programs are written so they can be compared to each other easily. This does not necessarily mean that they have been written in the best format for the language used. For example, since Pascal is relatively inflexible, the machine language and C programs have a very Pascal-like "bottom-up" format in order to keep the functions parallel.

Since all three programs make extensive use of routines in ROM, they run at roughly the same speed. They should be studied carefully and used as models for more complex DeskTop applications.

If you diligently typed in one of the model program listings, successfully compiled it, and were mildly impressed with the results, don't give up now. These model programs were intentionally created as skeletons. They form the basic parts of all DeskTop applications.

The chapters that follow discuss key portions of the sample programs in greater detail. They show how to add more pull-down menus, custom windows, dialog and alert boxes, and special dialog controls. With a little imagination and this book at your side, you're on your way toward a rewarding programming experience.

Chapter 7

Memory Management

Many Apple IIGS programmers have their roots in earlier Apple II computers. Perhaps you're one of them. If so, you are well aware of the anarchy that prevailed in the 64K RAM Apple II. Apple had cleaned up the neighborhood when the memory-management system was created

for the Apple IIGS. It was something that had to be done.

This chapter is about memory management. It may sound like a dry subject, but it really isn't. In fact, compared to the jungle-gym memory management of earlier Apple II computers, the designers of the Apple IIGS have blessed the programmer with a memory-management system that's reliable, easy to manage, and easy to program.

New Rules

Imagine an Apple II with eight megabytes of RAM (128 times more memory than a 64K Apple II) and no sensible way of managing it all. Programs would overwrite each other, and there would be no way to locate lost data, which might be intact but as irretrievable as a needle in a hayfield. Before long, memory would be as packed with as much useless information as a poorly managed bookstore. A horrific thought. But thanks to the Memory Manager built into the IIGS, programs can coexist in peace for the first time in Apple II history.

This is a radical departure from the programming environment of older Apple IIs. If you're moving up to an Apple IIGS from a IIe (or IIc or II+), you're in for a surprise. Gone are the days when a program grabbed a hunk of memory for its own purposes.

With the Memory Manager in charge, memory blocks are allocated to applications that request them. Memory blocks can be any size, and they can contain any type of information. But a program must specifically ask for a block of memory or risk the complete destruction of any space it arbitrarily claims.

A memory block may be located anywhere in RAM. It is very rare for a program to ask for a block of memory that always resides at a fixed address in the machine. In fact, it's considered sloppy programming if your application cannot deal with memory blocks that move around in the Apple IIGS. The memory blocks that the Memory Manager hands out will not always live at the same address in the computer, and there's a good reason for this.

As more and more applications reside inside the computer at the same time, their impact on memory usage will vary. Some programs might require a small portion of RAM for temporary usage and then throw it away when it's no longer needed. Other programs might require memory blocks that could be considered permanently reserved. And still other programs may require great amounts of memory. Managing that memory without the assistance of the Memory Manager would be a big headache.

So, imagine having hundreds of small memory blocks scattered throughout your computer's memory. Then imagine that your application needs a large, contiguous piece of RAM, but an unused area of memory that size doesn't exist. If you couldn't move the smaller blocks, rearranging them to make room for the one large block, the program would crash. With this kind of demand on RAM, things can get messy fast if memory blocks are not allowed to be moved.

Figure 7-1. Memory Blocks Distributed All Over Memory in a Random Dispersal

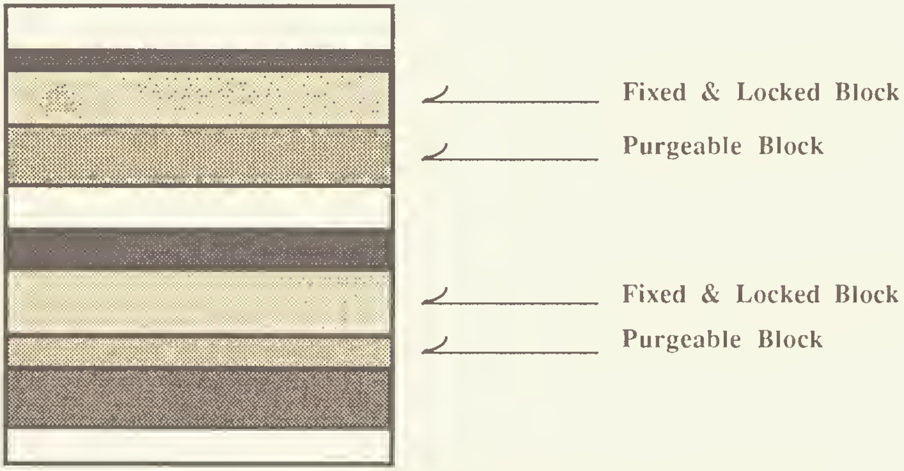
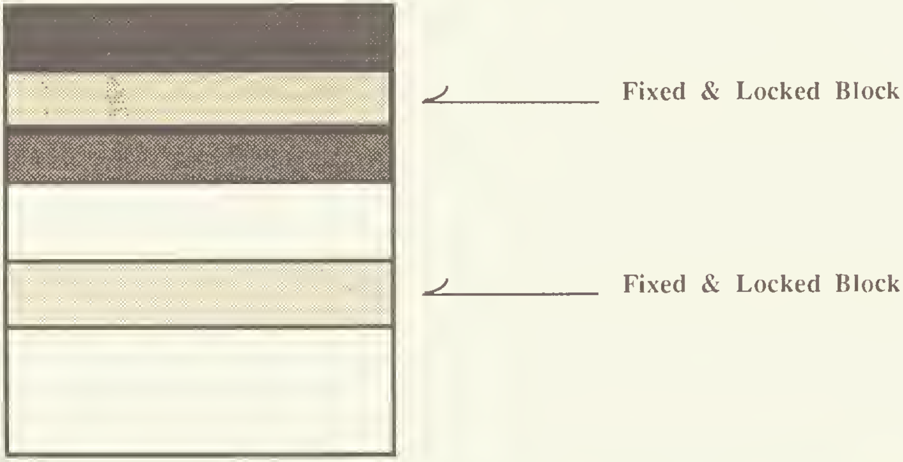

Fixed & Locked Block
Purgeable Block
Fixed & Locked Block
Purgeable Block

Figure 7-2. Memory Blocks After Reorganization by the Memory Manager


Fixed & Locked Block
Fixed & Locked Block

Fortunately, part of the Memory Manager's job is reorganizing memory blocks. It shuffles movable blocks around in an efficient and resourceful manner. This is done by removing blocks that are flagged as unused and then sliding blocks around in order to fill any gaps. The effect is that the landscape inside the computer is kept neat and orderly.

Don't let this worry you. It's possible for an application to request a block of memory that will reside at a fixed location, if you want it. However, the odds of the Memory Manager denying your request are higher because that space might already be reserved by another program.

Of course, if blocks of memory can be allowed to move about, seemingly at will, there must be a way to keep track of where they are.

Getting a Handle on Memory Blocks

Since a memory block can move around inside the computer, it is referenced by a handle. You'll see handles used with anything that moves about or that doesn't have a specific, given, or constant location, such as memory blocks, records, structures, and so on. Handles are simply long-word pointers to an address stored in memory, and they're used frequently in programming the Apple IIGS.

In the case of memory blocks, a handle points to a location in memory that contains a list of items. This list is also referred to as the *memory-block record*. For example, the first item in the list is a long-word address containing the actual location of the memory block's data in memory. The other items will be discussed later in this chapter.

Recall that a memory handle is a pointer. It points to a list of items. The first item in the list is an address which points to the location in memory where the memory block lives. This can be confusing.

If the memory block is moved, the only thing that changes is the address in the memory-block record. The handle still points to the same structure. Your program won't need to adjust anything if it's working with the handle correctly to begin with.

Suppose that you have a friend whose name is Kitty. On a page in your address book, you have recorded her name, address, birth date, and dozens of other pieces of information about her.

Kitty has a problem: She is always being evicted from her apartment. Whatever other information you have about Kitty is always the same: She never changes her hair color, her birth date, her parents, or her telephone number. Only her address. The line in your address book where her address is written is the only thing you need to change in order to keep up to date on Kitty. That much-erased line in your address book is analogous to the memory-block record.

Starting the Memory Manager

Just as its name implies, the Memory Manager is responsible for keeping the computer's RAM neatly organized. This is done by tagging with an identification number each chunk of memory owned by an application. Whenever the Memory Manager is called upon for moving, purging, or manipulating a block of memory, your program must identify its piece of RAM. This is done by passing along an identification value when calling the Memory Manager.

Even the space that your program occupies is branded with its own identification number. The ID of your program is obtained when the Memory Manager is started.

The following examples show how a program obtains its own ID. This is typically one of the first calls an application should make.

In machine language:

```
pha                         ;word result space
_MMStartUp                  ;start the Memory Manager
pla                         ;pull Master User ID
sta           UserID        ;and save it
```

In Pascal:

```
UserID := MMStartUp;
```

In C:

```
UserID = MMStartUp( );
```

These samples demonstrate the steps involved in starting the Memory Manager in 65816 machine language, Pascal, and C. The UserID, declared as a 16-bit unsigned integer, is a unique identifier that belongs to your application. It should always be saved for later use.

User ID Numbers

The ID value returned by MMStartUp is your program's master User ID. It references the space your application takes up in memory. The User ID is used when shutting down both your program and the Memory Manager.

The ID value consists of 16 bits, grouped into three parts, or fields. Bit positions within the word value represent the different fields:

Field:	Type ID				Aux ID				Main ID							
User ID:																
Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

The Type ID field occupies bits 12–15. Type ID identifies the class of software the User ID belongs to. It may be one of 11 values:

Value	Class of Software
$0	Memory Manager
$1	Application
$2	Control program
$3	ProDOS
$4	Tool set
$5	Desk accessory
$6	Runtime libraries
$7	System Loader
$8	Firmware
$9	Tool Locator
$A	Setup file
$B–$F	Undefined

The Auxiliary ID field occupies bits 8–11. This field is initially set to 0, but you can manipulate it to create up to 16 different sub-ID's for your program. For example, to set bit 8 (the least significant bit of the Aux ID field), the following can be done.

In machine language:

```
lda   UserID          ;Get the User ID . . .
ora   #%100000000     ;. . . and set bit 8
sta   MemID           ;Save the new ID
```

In Pascal:

```
MemID := UserID + 256;
```

In C:

```
MemID = UserID | 256;
```

This should be done before an application requests memory from the Memory Manager. An auxiliary ID value is used rather than the program's User ID. The memory allocated can then be categorized by your program as an example of using this field. Otherwise, if you don't want to get that detailed, you can ignore the Aux ID field. But it's there if you need it.

The Main ID field occupies the lower eight bits of the User ID returned from the Memory Manager. This is a unique number assigned to your program's User ID by the Memory Manager. Your User ID is what makes your program special. It makes your program different from any others that are running in the machine. The lower eight bits of your User ID should never be altered. To do so would be like changing your own fingerprints.

Asking for Memory

When a ProDOS 16 application is launched, it is given its own 64K bank of memory to live in. It also has its own direct page and stack. If the program requires memory outside its code space for storage, it must call the Memory Manager's NewHandle function to request a block of memory.

This 65816 code segment calls the NewHandle function in order to request a 256-byte buffer in bank $00 of the computer:

```
pha                         ;long word result space
pha                         ;
pushlong      #$100         ;push size of requested block (one page)
pushword      MemID         ;push a Memory ID (made from the User ID)
pushword      $C005         ;Attribute bits (discussed later)
pushlong #0                 ;Location of block in memory (not used)

_NewHandle                  ;call the NewHandle function
```

This call requires four input parameters (and result space when called from machine language) in order to work:

Value	**Parameter Description**
Long word	Size of the memory block needed
Word	An ID value to assign to this block
Word	Attributes (discussed later)
Long word	Location of the block (if applicable)

Value	Parameter Description
Size	The size of a memory block can be anything from zero bytes to whatever free memory space there is left in the machine.
ID	As described earlier in this chapter, each memory block allocated to a program must be identified by an ID number.
Attributes	The attributes of a memory block determine certain characteristics about it (where it can reside, if it can be moved, and so on). Attributes are very important. They will be discussed in detail later in this chapter.
Location	If the memory is to reside at a fixed location in memory, this long-word value determines the address requested.

NewHandle doesn't return a pointer to the location of the requested block of memory. Rather, it returns a handle to that block. The handle will be waiting to be pulled from the stack after this call is made from machine language—which is an important detail to remember.

The handle references the memory-block structure just allocated. Within that structure is the actual address of the memory block. That address is obtained in the following manner:

```
pla                    ;get low-order word of the handle
plx                    ;get high-order word of the handls
sta   0                ;build a long pointsr at location $00
stx   2
sta   TheHandle        ;and save the address for later
stx   TheHandle+2      ;(might be used for disposal)
```

The handle has been pulled from the stack and stored in four bytes from memory locations $00–$03. A copy has also been stored in TheHandle, a four-byte storage area within the program. By putting the value returned by NewHandle at location $00, a long-address pointer is created. This can be referenced indirectly in order to fetch values in the memory block's record.

```
lda   [0]              ;get 16-bit addrsss of the memory block
sta   BlockAddr        ;. . . and save it
ldy   #2               ;index passed the first word . . .
lda   [0],y            ;. . . then get the bank of ths msmory block
sta   BlockAddr+2      ;and save it
```

The address contained in the four-byte storage area named BlockAddr is the location of the 256-byte page of memory that was allocated with NewHandle. In fact, due to the location and attributes of this memory block, it could be used as direct-page space by a tool set.

Since direct pages reside in bank $00 only, the high word of the address need not be retrieved from the memory handle record. The most significant word of the address is assumed to be 0. Example:

```
lda   [0]           ;get the direct page address
sta   DPageAddr     ;. . . and save it
```

DPageAddr would simply be a two-byte storage area in your program.

Using NewHandle in Pascal and C is far easier. In Pascal, the following is used to obtain memory for direct-page space:

```
TheHandle := NewHandle($100, MemID, $C005, Ptr(0));
DPageAddr := LoWord(TheHandle^);
```

These statements are identical in operation to the machine language example listed earlier. The four parameters in the above example that constitute the NewHandle requirements are block size requested ($100), an ID (MemID), attributes ($C005), and the block's address (0).

The following illustrates grabbing a memory handle using C:

```
TheHandle = NewHandle(0x100L, MemID, 0xC005, nil);
DPageAddr = (int) *(TheHandle);
```

These statements are identical to the machine language and Pascal examples.

The Memory-Block Record

The memory-block record is one of those things you really don't need to know about in order to program the Apple IIGS. The structure and manipulation of memory handle records is not part of the regular programmer's repertoire. In fact, the only time you would examine a record is to locate a memory block's true location in memory. And the purpose of having a Memory Manager is to avoid that.

Each block of memory allocated by the Memory Manager has a corresponding record. (Recall that the record is what the handle points to.) The structure of this record consists of six fields that give the memory block's address in memory and provide additional information about the block.

The long-word value (handle) returned by NewHandle is the address in memory where the memory block's record is stored. This record is 20 bytes long, and it contains the following information:

Size	Contents
Long word	Address of the block
Word	Attributes
Word	Owner's User ID
Long word	Size of the block
Long word	Pointer to the next handle record
Long word	Pointer to the previous handle record

The first four fields are copies of the parameters used when the NewHandle call was first made. See the previous section for details.

The last two items require further explanation.

In order to keep track of these handle records, the Memory Manager uses a set of *next* and *previous* record pointers to create a linked list. The first long-word pointer points to the next 20-byte memory handle record, while the second pointer points to the previous record. This allows handle records to reside in any order throughout the computer's memory, yet they can be referenced in order due to their link fields.

Block Attributes

When NewHandle is used to allocate a block of memory, you must decide how that block should be treated by the Memory Manager. For example, should the block be allowed to move around? Does it have to be aligned on a 256-byte page boundary (which speeds up some processes)? Can it reside in special memory banks? You'll have to consider these points, and more, when allocating a new handle. Time to think.

Block attributes are assigned by the programmer before the NewHandle function is called. The attributes parameter is a word value and consists of 16 bits of information:

Bit	Meaning If Set (Made Equal to 1)
0	Block must reside in a particular memory bank
1	Block must reside at a particular address
2	Block must be page-aligned
3	Block can reside in special memory banks
4	Block cannot cross a bank boundary

Bit	Meaning If Set (Made Equal to 1)
5	Reserved
6	Reserved
7	Reserved
8	Purge level (low bit)
9	Purge level (high bit)
10	Not used (0)
11	Not used (0)
12	Not used (0)
13	Not used (0)
14	Block is fixed (cannot move)
15	Block is locked (fixed and unpurgeable)

Each bit position represents a specific attribute describing the memory block to be allocated. Setting a bit asserts that attribute.

Bit 0	Specifies whether the block should reside in a particular bank of memory in the computer. For example, if your application required a memory block that must reside in bank $05, you would set this bit.
Bit 1	Specifies that the block must live at a particular address in memory. Memory blocks that reside at fixed addresses should also have bit 14 set (which means they cannot be moved).
Bit 2	Causes the block of memory to reside on a page boundary. A page is 256, or $100, bytes of RAM. The first page boundary is at location $0000 in a bank. The next page is at location $0100. The next page would be at location $0200, followed by $0300, and so on, all the way up to $FF00, the last page boundary in a bank.
Bit 3	Determines if a block can reside in the special memory banks $00, $01, and $E0 and in bank $E1. These banks are used by the Mega II (Apple IIe–emulation) mode of the Apple IIGS when an application runs under the 8-bit version of ProDOS. If you create a memory-resident application for the Apple IIGS, such as a desk accessory, it cannot reside in special memory. In native (16-bit) mode, bank $00 is used mainly by DeskTop applications for direct-page space.
Bit 4	Tells the Memory Manager if the block can cross from one bank to the next in the computer. For example, a $2000-byte block, living at location $03FE00 could cross over into bank $04 if bit 4 was not set.

Bits 5–7	Reserved and should not be set.
Bits 8 and 9	Classify the purge level of a memory block. Because there are just two bits, four unique settings can be assigned (2 × 2 = 4):

Value	**Meaning**
0	The block cannot be purged
1	Very low purge level
2	Moderate purge level
3	Very susceptible to purging

Blocks are purged when the Memory Manager is called to compact memory and clean house. As you can see, blocks with the highest nonzero purge levels are purged first.

Bits 10–13	No use at this time.
Bit 14	Fixes a block in memory so it cannot be moved.
Bit 15	Used to lock a memory block. Locking causes the block to become immovable and unable to be purged, regardless of the settings of bits 8, 9, and 14.

The Memory Manager tool set provides functions for changing the attributes of a block after it has been allocated with NewHandle. They are the following:

Function	**Description**
HLock	Locks a memory block referenced by its handle
HLockAll	Locks all memory blocks referenced by a User ID
HUnLock	Unlocks a memory block referenced by its handle
HUnLockAll	Unlocks all memory blocks with a certain User ID
SetPurge	Sets the purge level of a block referenced by handle
SetPurgeAll	Sets the purge level for all blocks with the same ID

The summary at the end of this chapter lists the parameters for these functions. Note that *COMPUTE!'s Mastering the Apple IIGS Toolbox* provides parameter descriptions for the entire Apple IIGS Toolbox.

Removing Memory

Memory is removed by eliminating memory handles (the same handles that were obtained by the NewHandle function). When a handle is removed, the Memory Manager is allowed to make available the space that its memory block took up in the computer.

Any handles allocated by your application should be removed as soon as they are no longer needed. This will make the memory they occupy free for use by other applications. Handles can be removed in a number of ways using the Memory Manager tool set.

The most straightforward method of removing a memory block is to use the DisposeHandle function. Your application pushes the handle's value onto the stack and then DisposeHandle is called. The memory block and its allocated handle are removed from the system instantly.

In machine language:

```
pushlong TheHandle      ;push the handle on the stack
_DisposeHandle          ;and now dispose of it
```

The same example in C or Pascal:

```
DisposeHandle(TheHandle);
```

If you have allocated multiple handles with a single identification value, your application can take a shortcut by using the DisposeAll function. DisposeAll will remove all handles associated with a particular ID number. For example, in C or Pascal:

```
DisposeAll(MemID);
```

Your programs should never call DisposeAll with the master User ID returned by MMStartUp. This would cause the memory space that a program occupies to be freed, which might result in a system crash.

Another approach to freeing a block is to set its purge level to the highest setting (3). This would cause the Memory Manager to dispose of your block the next time it was called to compact memory (CompactMem). Note, however, that the handle remains allocated and will have to be removed eventually.

When a handle is purged, the block allocated to this handle is freed, but the handle is kept alive. The address of the block in the memory block record is set to $0000000 (a long word of 0). This tells the Memory Manager that the handle is valid, but does not have a block allocated to it. This would allow you to reallocate (ReAllocHandle) a memory block at a later time without having to use NewHandle to create a brand new one. It is understood that if purging does not dispose of the handle, your application will still need to do so before quitting.

Chapter Summary

The following Toolbox functions were discussed in this chapter. Also included are a few of the popular Memory Manager functions.

Function: $0202
Name: MMStartUp
Starts the Memory Manager
Push: Result Space (W)
Pull: UserID (W)
Errors: $0207
Comments: One of the first calls made by an application.

Function: $0302
Name: MMShutDown
Shuts down the Memory Manager
Push: UserID (W)
Pull: nothing
Errors: none
Comments: Make this call when your application is finished.

Function: $0902
Name: NewHandle
Makes a block of memory available to your program
Push: Result Space (L); Block Size (L); UserID (W);
Attributes (W); Address of Block (L)
Pull: Block's Handle (L)
Errors: $0201, $0204, $0207

Function: $0A02
Name: ReAllocHandle
Reallocates a purged block with new parameters
Push: Block Size (L); UserID (W); Attributes (W);
Address of Block (L); Old Block's Handle (L)
Pull: nothing
Errors: $0201, $0203, $0204, $0206, $0207

Function: $0B02
Name: RestoreHandle
Reallocates a purged block using original parameters
Push: Old Block's Handle (L)
Pull: nothing
Errors: $0201, $0203, $0206, $0208
Comments: Uses same parameters of original block (unlike function $0A which allows the parameters to be reset).

Function: $1002
Name: DisposeHandle
Deallocates a block and releases its memory
Push: Block's Handle (L)
Pull: nothing
Errors: $0206
Comments: The block is deleted regardless of its locked status or purge level.

Function: $1102
Name: DisposeAll
Releases all blocks associated with a UserID
Push: UserID (W)
Pull: nothing
Errors: $0207
Comments: Ruthless.

Function: $1202
Name: PurgeHandle
Purges a block of memory
Push: Block's Handle (L)
Pull: nothing
Errors: $0204, $0205, $0206
Comments: The block must be purgeable and unlocked. The block's handle is not deallocated by this call.

Function: $1302
Name: PurgeAll
Purges all blocks associated with a UserID
Push: UserID (W)
Pull: nothing
Errors: $0204, $0205, $0207
Comments: The blocks must all be purgeable and unlocked.

Function: $1B02
Name: FreeMem
Returns memory available for programs
Push: Result Space (L)
Pull: Integer Value (L)
Errors: none
Comments: Returns the total number of bytes in memory, not counting ramdisks or other allocated blocks.

Function: $1C02
Name: MaxBlock
Returns memory available to programs
Push: Result Space (L)
Pull: Integer Value (L)
Errors: none
Comments: Returns the largest free block in memory.

Function: $1D02
Name: TotalMem
Returns total RAM in the System
Push: Result Space (L)
Pull: Integer Value (L)
Errors: none
Comments: Returns all RAM in your Apple IIGS, including the basic 256K, any ramdisks, and so on.

Function: $1F02
Name: CompactMem
Compacts memory
Push: nothing
Pull: nothing
Errors: none
Comments: Performs memory garbage collection, purging purgeable blocks and reorganizing memory. Don't do this during an interrupt.

Function: $2002
Name: HLock
Locks and sets a specific handle to a purge level of 0
Push: Handle (L)
Pull: nothing
Errors: $0206

Function: $2102
Name: HLockAll
Locks and sets all handles associated with a specific UserID to a purge level of 0.
Push: UserID (W)
Pull: nothing
Errors: $0207

Function: $2202
Name: HUnLock
Unlocks a block of memory
Push: Handle (L)
Pull: nothing
Errors: $0206

Function: $2302
Name: HUnLockAll
Unlocks all blocks of memory associated with a specific UserID
Push: UserID (W)
Pull: nothing
Errors: $0207

Function: $2402
Name: SetPurge
Sets the purge level of a given block
Push: New Purge Level (W); Handle (L)
Pull: nothing
Errors: $0206
Comments: Only the lower two bits of the word pushed are significant.

Function: $2502
Name: SetPurgeAll
Sets the purge level for all blocks associated with a given UserID
Push: New Purge Level (W); UserID (W)
Pull: nothing
Errors: $0207

Function: $2B02
Name: BlockMove
Copies a block of memory from one address to another
Push: Source Address (L); Destination Address (L); Length (L)
Pull: nothing
Errors: none

Memory Manager Tool Set Error Codes

$0201	Unable to allocate block
$0202	Illegal operation on an empty handle
$0203	Empty handle expected for this operation
$0204	Illegal operation on a locked or immovable block
$0205	Attempt to purge an unpurgeable block
$0206	Invalid handle given
$0207	Invalid User ID given
$0208	Operation illegal on block-specified attributes

Chapter 8

Pull-Down Menus

In menu-driven programs not too many years ago, the computer's monochrome screen would clear and a long list of menu items, usually numbered, marched down the display:

```
MAIN MENU OPTIONS:

1. GO TO MENU 2
2. GO TO MENU 3, SUB MENU C
3. GO TO MENU 5 AND STAY THERE
4. DO MAIN MENU OPTION 7
5. PRETEND TO GO TO MENU 7 BUT GO TO MENU 6 INSTEAD
6. GIVE ME THE BREAKFAST MENU
7. DO MAIN MENU OPTION 4
8. JUST GET ME THE CHECK

ENTER YOUR SELECTION (1-8):
```

Pressing a number would erase the old menu and would likely unravel yet another screenful of menu items. Sometimes this would go on through several levels, before anything could get done. Mobility in this environment was like jogging blindfolded—with shackled legs.

With a DeskTop environment however, the user can see all the possible menus at once. Their titles are positioned horizontally across the top of the screen. Navigating through these menus requires little instruction. They are intuitive and are becoming commonplace in the computer world. Just about everyone has had exposure to them.

The Two Managers

Programmers who have written interactive software know that when life is made easier for the user, it usually means the opposite for the programmer. Creating user-friendly software requires hard work. While this is generally true for applications in other environments, things couldn't be sweeter for the Apple IIGS programmer. All the credit goes to the Menu and Window Managers.

As its name implies, the Menu Manager is responsible for maintaining the lists of numerous commands and functions a program may contain. It takes care of shuffling menus around, drawing them on the screen, and interacting with the user while selections are made with the mouse or keyboard.

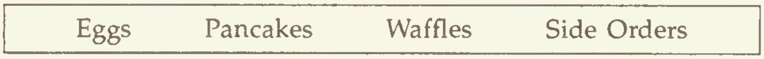
What does the Window Manager have to do with menus? A vital part of the Window Manager is the TaskMaster. The purpose of the TaskMaster is to watch for menu events that occur on the DeskTop and to handle them appropriately. It relieves the programmer of those bothersome details. However, if an application requires custom event handling, the TaskMaster can be bypassed altogether.

Organizing Menu Items

Organization of functions and subroutines is an essential step in creating any new program. The same applies to creating pull-down menu items. For example, a coffee-shop menu is grouped into sections such as Eggs, Pancakes, Waffles, and Side Orders. This makes it easier for the diner to locate a particular item.

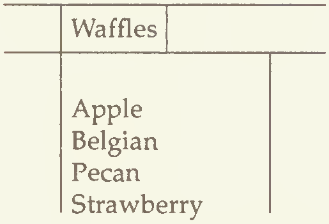
In a DeskTop program, the Main Menu of yesteryear's application is replaced by the System Menu Bar at the top of the screen, as shown in Figure 8-1.

Figure 8-1. Breakfast Menu Bar



Within each menu are menu items. For example, the third menu, Waffles, might include four items, shown in Figure 8-2.

Figure 8-2. Waffle Menu



For the user's sake, items in a pull-down menu should be related to the title of the menu. This falls into the department of Apple's Human Interface Guidelines (see Appendix A). The guidelines were created to help the programmer decide where certain commands should go, how they should be named, and so on.

As an example, commands that open and close files, save changes to disk, create new files, and interact with the printer, are found in the File menu on the System Menu Bar. The command to quit a program is also in the File menu. Practically all DeskTop programs have a File menu so long as a means exists for quitting the application.

Once an application's commands are organized into menus, the programmer must decide which, if any, should have keyboard equivalents. Keyboard equivalents are awarded to commands used most often. Applications relying heavily on keyboard input, such as word processors, ought to provide the user with as many key equivalents as possible. On the other hand, people tend to make menu selections using the mouse while working with drawing or painting programs, which makes it less important that graphics program menu items have keyboard-equivalent commands.

According to the Human Interface Guidelines, created by Apple's Bruce Tognazzini (lovingly known as "Saint" Tognazzini), some keyboard command characters should be reserved for certain functions in order to maintain consistency from one DeskTop application to the next.

Table 8-1. Command Key Equivalents

Key	Command
C	Copy
O	Open
Q	Quit
S	Save
V	Paste
X	Cut
Z	Undo

The letters listed in Table 8-1 are commonly reserved for the listed functions.

Keyboard equivalents are shown to the right of a menu item, preceded by the Open Apple symbol. On the Macintosh, they are preceded by the clover-leaf (Command key) symbol.

The placement of items on the menu bar is also discussed in the Human Interface Guidelines. The menus are positioned on the menu bar starting with the Apple menu (also called the New Desk Accessory menu) at the left side. Following that comes the File menu. And if the application manipulates text or graphics, usually an Edit menu follows. Consult Appendix A for other reserved menu titles suggested by the guidelines.

Designing a Menu

The Menu Manager works with strings of characters in order to build a menu and create its contents. A list of these strings is passed to the Menu Manager via the NewMenu Toolbox function. In machine language, C, or Pascal, the data for a menu list can be created by defining text-string constants.

These strings must remain in memory for as long as the menu bar is present. Machine language programmers should not reuse the space occupied by these strings, and C programmers should define the strings as global, static text.

A menu list consists of three parts:

- A title
- The menu items
- The end of menu marker

Additionally, each item of a menu, and its title, are tagged by a unique identification number. The ID number of a menu title is useful only to the Menu Manager. The ID number of a menu item is used by the application when the user selects a menu item.

Figure 8-3 shows a sample menu list.

Figure 8-3. Menu List

```
>> Waffle  \N3          ← Menu Title
--Apple \N256
--Belgian \N257         ← Menu Items
--Pecan \N258
--Strawberry \N259
>                       ← End of Menu
```

Each line in the list begins with two unique characters. The exception is the last line which requires just one character.

The first line in the list describes the title of the menu. It begins with two letters, numbers, or symbols. Following these characters is the menu title. Incidentally, the title is usually surrounded by one or more spaces to provide padding between the other titles on the menu bar. The backslash character (\ ) signals the end of the title and the beginning of the special characters. Therefore, a backslash cannot be part of the menu's name. The special characters further describe the menu item.

The commercial *at* symbol (@) is used to produce the colored Apple logo used for the Desk Accessory menu. It must be the only character in the title, with no surrounding spaces.

You can also specify the @ sign for any other menus you may have. However, only the Apple logo will appear as long as there is no other text along with it.

The strings that follow the title line make up the list of items in this menu. Each line starts with the same two characters, which can be any characters, except the two that begin the menu's title line. The name of the menu item follows, and then finally, a backslash signals the start of the special characters.

A special menu item, called a dividing line, can be placed into the menu by using a single hyphen as a menu item. Its purpose is to divide members in an item list. Dividing line items should be dimmed (see below) so that they cannot be chosen as a legal menu item.

The very last line of the menu list consists of a single character. This character must be different from the characters that start the menu item lines. However, it can be the same character that begins the title line, as shown in Figure 8-3 above.

Each line in the list, except for the very last, ends with a carriage return ($0D) or a null character ($00). This tells the Menu Manager that the end of the line has been encountered and it is allowed to proceed to the next line.

The special characters that follow the backslash have the following functions:

Character	**Does This**
*	Defines the command key equivalents
B	Draws the menu item's text in boldface
C	Places a character in front of the item name
D	Dims and disables the menu or menu item
H	Indicates that a two-byte hexadecimal ID number follows
I	Draws the menu item's text in italic style
N	Indicates that a decimal ASCII ID number follows
U	Underlines the menu item's text
V	Places a dividing line between this item and the next
X	Activates color replace for highlighting

These characters can be upper- or lowercase. Two characters must follow the * for keyboard equivalents. They are used to specify the case sensitivity of the command letter. For example:

*Bb	Both *B* and *b* are accepted
*BB	Only uppercase *B* is accepted
*bb	Only lowercase *b* is accepted

Similarly, using *?/ would allow the user to press the Open Apple key and either the slash or question mark (shift-slash), for example, to execute a Help command.

The key equivalent will be displayed on the menu after an Apple symbol. Also, only the first letter after the * is displayed on the menu, though both keys will work.

The *B*, *I*, and *U* special characters, which stand for boldface, italic, and underline type styles, respectively, are used to enhance the display of text items. They may or may not be available for use depending on the system font.

The letter *C* places a character before the item's text. Typically, this is used to mark the item with a special character, such as the following:

Character	**ASCII Value**
Check mark	18 ($12)
Diamond	19 ($13)
Open Apple	17 ($11)
Solid Apple	20 ($14)

For example, to place a check mark before a menu item, the following string would be defined in a machine language source code file:

```
dc     c'--Checked Item\C',il'18',c'N256V',il'0'
```

More information on creating the menu strings from assembly language is covered in the next section.

D is used to dim and disable an item. The item appears in a dimmed state and cannot be chosen. If the menu itself is disabled, every item in that menu will be dimmed and disabled.

H and *N* allow a menu or item to be assigned an ID number. When *H* is used, it's followed by a two-byte hexadecimal value in low-byte/high-byte order. If *N* is used, it is followed by a string of decimal characters. Not every menu item requires an ID, and only certain IDs are used as shown in the following chart:

Menu IDs	**Description**
0	Used internally
1–65534	Used by an application's menus
65535	Used internally

Item IDs	**Description**
0	Used internally
1–249	Used by desk accessory items
250	Undo
251	Cut
252	Copy
253	Paste
254	Clear
255	Close
256–65534	Used by an application's menu items
65535	Used internally

Identification numbers don't have to be sequential or defined in any order. They have to be unique, but only if they are enabled. Machine language programmers will want to assign menu item IDs starting with 256 and work upwards, not skipping over any values. (The reason for this is discussed later.)

V is used to draw a dividing line between two items, across the entire width of the menu. It does not take up a line in the list of items, as the hyphen character does. (See above.)

X uses color-replace mode that affects the way a menu is highlighted when it is chosen. When a colored menu is selected with color replace activated, the colors will remain the same. For example, in the Apple menu, the X option should be specified. If not, the Apple character will appear gray on a black background, rather than colored on a black background. For ordinary menus, the X option need not be specified.

Creating Menu Strings

When using the *APW* assembler, string constants are defined using the DC (Define Constant) directive:

```
Menu3   dc    c‘>> Waffle \N3’,i1‘0’
        dc    c‘--Apple\N256*Aa’,i1‘0’
        dc    c‘--Belgian\N257*Bb’,i1‘0’
        dc    c‘--Pecan\N258*Pp’,i1‘0’
        dc    c‘--Strawberry\N259*Ss’,i‘0’
        dc    c‘>’
```

Each line ends with a single zero byte. ID numbers are assigned using the special character *N*. However, the *H* character could have been used:

```
dc    c‘--Apple\H’,i‘256’,c‘*Aa’,i1‘0’
```

Why use *H* when *N* will do? Because it saves a byte and is easier for the Menu Manager to parse. Unfortunately, it makes the source code look messy.

Things are done a bit differently using the C language. The Waffle menu could be defined using static text strings as follows:

```
char *Menu3[] = { “>> Waffle \\N3”,
                  “--Apple\\N256*Aa”,
                  “--Belgian\\N257*Bb”,
                  “--Pecan\\N258*Pp”,
                  “--Strawberry\\N259*Ss”,
                  “>” };
```

Since the C compiler uses the backslash character for various purposes, it must be entered twice in a row in order to insert one backslash into a string of text. And since C strings by definition end in a null byte, the end-of-line terminator will be inserted automatically at compile time.

> An alternative method to define text strings is to use C's in-line assembly feature to define the strings with 65816 instructions. Or you could write an external program in machine language that is linked with the C code later on.

For programmers using *TML Pascal,* menu strings must be defined as global string types. They are built at runtime using the CONCAT function:

```
Menu3 := CONCAT(‘>> Waffle \N3D\0’,
                ‘--Apple\N256*Aa\0’,
                ‘--Belgian\N257*Bb\0’,
                ‘--Pecan\N258*Pp\0’,
                ‘--Strawberry\N259*Ss\0’,
                ‘>’);
```

Notice how each line is terminated by the null, \0, escape sequence. Unlike C, Pascal strings are not automatically terminated by nulls, and, therefore, the programmer must provide them.

Installing a Menu

Before any Menu Manager functions can be called, the Menu Manager must be started. The Menu Manager, like a few other tool sets, also requires its own direct page. If you're not sure how to start up a tool set, or how to get direct page space, see Chapter 4 in this book, "About the Toolbox," and Chapter 7, "Memory Management."

Placing a menu into the System Menu Bar is a two-step process. First, all menu strings must be passed to the NewMenu function. NewMenu uses them to create an internal menu record. Once completed, NewMenu returns a handle to the menu record.

The second step involves inserting the menu record into the System Menu Bar by using the InsertMenu function. This is done by passing the menu handle, returned by NewMenu, to InsertMenu. It then places the menu at the desired position.

To accomplish this process from a machine language program, the following can be used:

```
         pha                     ;long-word result space
         pha                     ;long-word result space
         pushlong       #Menu3   ;point to Menu 3's strings
         _NewMenu                ;create the menu record . . .
;                                ;. . . whose handle is now on the stack
         pushword       #0       ;insert it before all other menus
         _InsertMenu
```

InsertMenu's two input parameters are the handle of the menu record and a position value that determines where on the menu bar the menu title will be inserted. If the position is 0, the menu will be the leftmost menu. Note how the menu record handle is kept on the stack for the call to _InsertMenu.

> The position argument, if 0, will insert the menu at the leftmost side of the menu, pushing any existing menus to the right. But if the position value is a Menu ID number, it instructs the Menu Manager to insert the menu after the menu referenced by that ID.

Creating and inserting a menu in C or Pascal is practically effortless when compared to machine language.

With C:

```
InsertMenu(NewMenu(Menu3[0]), 0);
```

With *TML Pascal*:

```
InsertMenu(NewMenu(@Menu3[1]), 0);
```

The two tasks can be taken care of with just one statement by embedding the NewMenu function within the InsertMenu function. This is a very common programming technique.

TML Pascal requires the *at* symbol in front of the Menu3 variable in order to reference its address in memory. Also, the data in Pascal strings starts with element 1, because element 0 is a count byte.

Drawing the Menu Bar

Even though a menu has been inserted into the menu bar, it does not appear on the screen. To cause the menu to appear, call the FixMenuBar function.

In machine language:

```
pha                 ;word result space
_FixMenuBar
pla                 ;returns the bar's height in pixels
sta   Height        ;(optional—you don't need to save it)
```

Or in C:

```
Height = FixMenuBar( );
```

The Height assignment is optional. A simple FixMenuBar( ) alone can be used.

In Pascal,

```
Height := FixMenuBar;
```

does the same, but the variable assignment (Height) is required.

FixMenuBar calculates the height of the System Menu Bar and vertical placement of menu items. This depends on the type of system font in use. If this function is not called, all the menu items will appear on top of each other, and the program will look peculiar.

Finally, when the menu records have been created, inserted, and fixed, the System Menu Bar can be displayed on the screen using the DrawMenuBar function.

With C, use

```
DrawMenuBar( );
```

Or with Pascal, use

```
DrawMenuBar;
```

To perform the equivalent with a machine language macro call, use

```
_DrawMenuBar
```

This function displays the titles of all your pull-down menus on the menu bar.

Using the TaskMaster

The easiest way to manage your menus is to let the TaskMaster do all the work. The TaskMaster takes over whenever the user does something to affect the menu-bar area. As an example, if the user clicks the mouse over a menu title, the TaskMaster calls the functions in the Menu Manager that draws the menu on the screen.

If the user begins to drag the mouse pointer down through a menu, TaskMaster calls the appropriate Menu Manager functions that allow the user to make a selection. TaskMaster also recognizes keyboard equivalents of menu items and treats them as if menu selections were made with the mouse.

Before TaskMaster is used, your application must provide an event record where TaskMaster places information. The event record consists of seven fields, structured in this manner:

```
EventRec     anop                  ;Event Record used by TaskMaster
What         ds     2              ;word
Message      ds     4              ;long word
When         ds     4              ;long word
Where        ds     4              ;long word
Modifiers    ds     2              ;word
TaskData     ds     4              ;long word
TaskMask     dc     i4'$1fff'      ;long word
```

The address of this record is passed to TaskMaster as one of its arguments.

Calling TaskMaster with machine language:

```
pha                           ;Result Space
pushword     #$FFFF           ;Event Mask (screen all event types)
pushlong     #EventRec        ;Point to Event Record
_TaskMaster
pla                           ;Get Event code
```

Calling TaskMaster with C:

```
Event = TaskMaster(0xffff, &EventRec);
```

After calling TaskMaster, a code is returned to your application. If its value is not 0, an event is pending. The application can continue to call TaskMaster until a nonzero code is reported. This is demonstrated by the following loop in Pascal:

```
REPEAT
    Event := TaskMaster($ffff, EventRec);
UNTIL Event <> 0;
```

When the user eventually makes a menu selection, TaskMaster returns control to your application, informing it that an event has occurred. If a menu item (other than a desk accessory) has been selected, TaskMaster returns an extended event code of $0011 (17 decimal). This is usually equated to the constant called wInMenuBar, as shown in this Pascal statement:

```
IF Event = wInMenuBar THEN DoMenu;
```

The lowercase *w* in *wInMenuBar* identifies it as a Window Manager constant. In *TML Pascal,* this constant is already defined as 17 for your application's use.

When menu event $11 has occurred, the menu number and menu item ID of the item selected can be obtained from the TaskData field in the Event Record.

Table 8-2 shows the contents of the TaskData field and how each word is referenced from machine language, C, and Pascal. Some real-life examples follow.

Table 8-2. TaskData Field

Language	**Low-Order Word Menu Item ID**	**High-Order Word Menu Number**
Machine language	TaskData	TaskData+2
C	EventRec.wmTaskData	EventRec.wmTaskData<<16
Pascal	LoWord(EventRec.TaskData)	HiWord(EventRec.TaskData)

To retrieve the menu selection in machine language:

```
lda   TaskData+2
sta   MenuSelected
```

To retrieve the menu selection in Pascal:

```
MenuSelected := HiWord(Eventrec.TaskData);
```

To retrieve the menu selection in C:

```
MenuSelected = EventRec.wmTaskData<<16;
```

The contents of the high- and low-order words of the TaskData field break down as follows:

Low-order word	The low-order word of TaskData holds the Menu Item ID of the selected item. For example, if the Pecan item in the Waffle menu were selected, the low-order word of TaskData would contain 258 (see Figure 8-3).
High-order word	The high-order word of TaskData contains the Menu Number. Again, if the Pecan item were selected, the high-order word of TaskData would contain 3.

Dispatching Item Handlers

Once the Item ID is known, as obtained from TaskData, the appropriate action can be taken by the program. Suppose that when the user has selected the Pecan item from the Waffle menu, you want the program to execute the PecanWaffle routine. C and Pascal programmers can use the SWITCH and CASE statements to do this.

The CASE statement example in Pascal:

```
CASE LoWord(EventRec.TaskData) OF
     256 :    AppleWaffle;
     257 :    BelgianWaffle;
     258 :    PecanWaffle;
     259 :    StrawberryWaffle;
END;
```

PecanWaffle is a previously declared procedure. It fulfills the user's request, perhaps by bringing up a dialog box asking if whipped cream is desired on the pecan waffle.

Dispatching the corresponding routine in machine language is done in one of two ways. The brute force method is to compare the item ID with an immediate value. If the two numbers match, a branch is made to the appropriate subroutine. Otherwise, the program continues to compare the ID with other immediate values.

A more elegant method, common among experienced machine language programmers, is to use the lower eight bits of the Item ID as an index into a table of pointers that point to the corresponding routines. It sounds more complex than it is. For example:

```
lda   TaskData       ;Get TaskData Item ID number
and   #$00FF         ;Discard upper 8 bits
asl   A              ;Double the value
tax                  ;Transfer to X as an index
jsr   (MTable,X)     ;Dispatch the proper menu item handler
```

If Pecan (item 258, the third menu item) were selected, the AND #$00FF instruction results in $02. This is multiplied by 2 using the ASL instruction which produces $04. That value is transferred to the X register to be used as an index.

> Using an index into a table of subroutines is one example of how useful numbering your menu items sequentially can be. The drawback is that during the cycle of development, you'll often move, reassign, insert, or change your menus as the program evolves. When this happens, renumbering menu items to keep them sequential can turn into a headache.

```
MTable  dc    i'AppleWaffle'          ;Item 256 (X = 0)
        dc    i'BelgianWaffle'        ;Item 257 (X = 2)
        dc    i'PecanWaffle'          ;Item 258 (X = 4)
        dc    i'StrawberryWaffle'     ;Item 259 (X = 6)
```

The JSR (MTable,X) instruction is known as an indexed, indirect jump to subroutine. The processor jumps to the two-byte address in MTable plus the value in the X register. Since X is 4, the subroutine PecanWaffle in the above table would be executed.

Unhighlighting the Menu's Title

During the dispatch of a menu item, the menu's title remains highlighted on the menu bar. This reminds the user that a menu item is being handled. When the service routine is finished, the menu's title should be inversed (unhighlighted). This is done with the HiliteMenu function.

In machine language:

```
puehword     #FALSE        ;Unhilite the menu title now
pushword     TaskData+2    ;Pueh TaskData Menu number
_HiliteMenu
```

With *TML Pascal*:

```
HiliteMenu(FALSE, HiWord(EventRec.TaskData));
```

And in C:

```
HiliteMenu(FALSE, EventRec.wmTaskData>>16);
```

The integer constant, FALSE, is defined as 0.

Keep in mind that unhighlighting a menu item is not automatic. You must do it manually after each menu item's function is completed.

Changing Menu Items

Not only are menu items an excellent way to initiate subroutines in an application, but they can also be used to toggle certain states (flags) in your program.

In a drawing program, for example, a check mark may appear next to the Ruler Guides item in the Tools menu. This would indicate that the rulers are in use. Should the user wish not to have rulers while painting (perhaps the artist is an impressionist), the Ruler Guides item could be selected from the Tools menu, which toggles the rulers off; the check mark would then disappear. But that doesn't happen by magic.

Assume that Ruler Guides has a Menu Item ID of 268. To place a check mark to the left of its name in the menu, the CheckMItem function is used:

```
pea   TRUE       ;TRUE: yes, check the item
pea   268        ;Item 268 (Ruler Guides)
ldx   #$320F     ;CheckMItem
jsl   $E10000
```

In C or Pascal:

```
CheckMItem(TRUE, 268);
```

Conversely, to remove a check mark or to make sure that there isn't one there, the same code can be used but with a FALSE value pushed to the stack instead of TRUE.

If your program has many menu items with check marks, it's best to create one procedure responsible for updating the checkmark state of all the items. An example in C:

```
UpdateCheckMarks( )
{
      CheckMItem(Rulers, 268);
      CheckMItem(BigBits, 270);
      CheckMItem(Clamps, 271);
      CheckMItem(WindowLocks, 273);
      CheckMItem(ColorMode, 281);
}
```

The integer (or Boolean) variables Rulers, BigBits, Clamps, WindowLocks, and ColorMode contain values representing true or false for the states of those items. If, in this drawing program, Rulers are turned on, but in order to use them Clamps and WindowLocks must be turned off, this C function would handle the correct toggling of Rulers:

```
ToggleRulers( )
{
      Rulers = !Rulers;                          /* Logical NOT toggle */
      if (Rulers)
            Clamps = WindowLocks = FALSE;
      UpdateCheckMarks( );
}
```

This is how the routine works:

- Toggle the current Rulers state to its opposite.
- If Rulers are now turned on (true), then make sure that Clamps and WindowLocks are turned off (false).
- Finally, update all the check marks according to the new states.

Another example of this technique, though not exactly similar to placing and removing the check mark, is the dimming of menu items, disabling them so that they cannot be selected. To disable a menu item, the DisableMItem function is used.

In machine language:

```
pushword   #256
_DisableMItem
```

In C or Pascal:

```
DisableMItem(256);
```

DisableMItem requires a menu item ID number as its argument. After the call is made, that menu item will be dimmed and not available for selection. To enable the item once again, the EnableMItem is used in a similar fashion:

In machine language:

```
pushword   #256
_EnableMItem
```

In C or Pascal:

```
EnableMItem(256);
```

Disabled menu items show up in a dimmed font in the pull-down menu, and the user of your program will not be able to select that item until it is enabled again.

Setting Menu Flags

Even though the Menu Manager has tools dedicated to one particular task, the SetMenuFlag function can perform the duties of three functions in one. SetMenuFlag works on an entire menu and affects all of its items. The following examples show what a typical call looks like.

In machine language:

```
PushWord     #MenuFlag     ;New menu flag value
PushWord     #MenuID       ;Menu ID number
_SetMenuFlag
```

In C and Pascal:

```
SetMenuFlag(MenuFlag, MenuID);
```

The values and attributes for the MenuFlag argument are expressed in Table 8-3. For example, using SetMenuFlag with $FFDF to invoke color-replace mode is the same as putting the special letter *X* in that menu's definition string.

Table 8-3. Values and Attributes of the MenuFlag Argument

MenuFlag	Description	Action
$FF7F	Enable	Menu becomes undimmed and its items selectable.
$0080	Disable	Menu becomes dimmed and its items not available.
$FFDF	Color Replace	Highlighting uses the color-replace method.
$0020	XOR Highlight	Highlighting uses the color XOR method.
$FFEF	Standard	Defines the menu as a standard type.
$0010	Custom	Defines the menu as a custom type.

Setting Item Flags

While SetMenuFlag (discussed in the previous section) reigns over entire menus, the SetMItemFlag function allows the attributes of a single menu item to be modified.

Table 8-4 provides a reference to the values that may be placed in the ItemFlag argument and their results.

Table 8-4. Values and Attributes of the ItemFlag Argument

ItemFlag	Description	Action
$FF7F	Enable	The item is enabled, selectable, and not dimmed.
$0080	Disable	The item is dimmed and disabled.
$FFDF	Color Replace	Highlighting uses the color-replace method.
$0020	XOR Highlight	Highlighting uses the color XOR method.
$0040	Underline	The item is drawn with an underline.
$FFBF	No Underline	The item is not underlined.

This is how a machine language routine that places a value in an Item Flag would look:

```
PushWord     #ItemFlag     ;New item flag value
PushWord     #ItemID       ;Item ID number
_SetMenuFlag
```

In C and Pascal:

```
SetMenuFlag(ItemFlag, ItemID);
```

Of course, the EnableMItem and DisableMItem functions should be used for enabling and disabling menu items just to keep your code looking clean and logical.

Menu Miscellany

The rest of the chapter deals with some of the minor details of working with the Menu Manager. Everything from changing the blink rate of a selected menu item to removing an entire menu is discussed in this section. This is where the fun starts.

Changing the Text Style

Menu items can appear in the standard text face or in special styles set by using the SetMItemStyle function. The normal system font can be displayed only in a bold style. However, the Toolbox has provisions for italic, underline, outline, and shadow styles when used with compatible fonts.

This brief table describes the effect of entering various values in the Style Word:

Style Bits	Style
0	Bold
1	Italic
2	Underline

Style Bits	Style
3	Outline
4	Shadow
5–15	Reserved

If a bit is set in the Style Word, it asserts that attribute. The following examples will set a bold style on the text of a menu item.

In machine language:

```
PushWord     #1      ;Bold
PushWord     #262    ;Item ID
_SetMItemStyle
```

In C and Pascal:

```
SetMItemStyle(1, 262);
```

To modify only one style bit without changing the others, use the GetMItemStyle function to return the current style, manipulate the appropriate bits, and then update the item with SetMItemStyle. This C language example sets a bold style to item #262 without changing any of its other style attributes:

```
Word Style;                     /* Style is an unsigned integer */

Style = GetMItemStyle(262);     /* Get the current style */
Style = Style | 1;              /* Logically OR with 1 */
SetMItemStyle(Style, 262);      /* Set the new style */
```

Or, the most compact form could be used:

```
SetMItemStyle(GetMItemStyle(262) | 1, 262);
```

Renaming a Menu Item

It's common to change the name of a menu item. In most cases, renaming an item draws a close relationship to using a check mark to show a certain state. For example, say you've written a communications program in which one of the items on a menu is Text Editor. By choosing this item, a user of your application is taken out of terminal mode and is placed into an editor mode. This would be an opportune time to rename that menu item, since Text Editor is no longer a valid choice: The user is already in it. Instead, that item could be renamed to Terminal Mode. By selecting this, the user could leave the editor and return to the terminal mode.

Changing the name of a menu item is quick and easy, as shown in these examples.

In machine language:

```
        PushLong    #NewName     ;Point to the new title
        PushWord    #262         ;Specify the item ID
        _SetMItem                ;Change the item's name
        rts

NewName dc    c'--Terminal Mode',i1'0'
```

In Pascal:

```
PROCEDURE NameMItem;
VAR
     NewName : String;
BEGIN
     NewName := '--Terminal Mode\0';
     SetMItem(@NewName[1], 262);
END;
```

In C:

```
SetMItem("--Terminal Mode", 262);
```

The SetMItem function requires two arguments:

- The address of a menu item string
- An integer that represents the ID of the item to rename

The string containing the new name is formatted just like a menu item: It begins with two starting characters (used only by the Menu Manager), and it ends with a null character.

Recall that strings in C always end with a null character. Therefore, there is no need to explicitly add one to the initialization string in the C example above.

SetMItem changes only the name of the item. All previous attributes—such as style, enabled or disabled states, and so on—are preserved. Even if the new item string contains a backslash (\ ) followed by special characters, only the name will be replaced. You can change attributes by using other Menu Manager functions discussed throughout this chapter.

Pascal users will undoubtedly want to use SetMItem's cousin, SetMItemName which is similar in syntax. The difference is that SetMItemName accepts a pointer to a Pascal string. Remember that strings in Pascal always start with a count byte. Here is an alternate Pascal example using SetMItemName:

```
SetMItemName('Terminal Mode', 262);
```

SetMItem is used to change the Save menu item in most Apple IIGS programs. After a file is opened, the Save item reads Save DOCUMENT, where DOCUMENT is the name of the file the user has opened. Simple string concatenation functions can be used in conjunction with SetMItem to accomplish this.

Although you've renamed the menu item, you're not done just yet.

When an item is renamed, the menu in which the item resides must adjust itself to the new width of the item, especially if it is longer than any of the others. This is done by using the CalcMenuSize function as demonstrated below.

In machine language:

```
PushWord    #0    ;New Width (0 = auto adjust)
PushWord    #0    ;New Height (0 = auto adjust)
PushWord    #2    ;The Menu's ID (not Item ID!)
_CalcMenuSize
```

C and Pascal:

```
CalcMenuSize(0, 0, 2);
```

CalcMenuSize, when used with nonzero arguments, can be used to set a menu's explicit height and width in pixels. If 0's are used, the Menu Manager will scan through the menu strings and automatically calculate the size of the menu, with room for checkmarks and Apple key equivalents. CalcMenuSize requires the ID of a menu as its third argument.

If the menu width is not resized, long menu item names will bleed right off the edge of the menu and into the DeskTop, which looks messy.

Renaming a Menu

It is far less common to change the title of a pull-down menu, but the Menu Manager will let you do it. The procedure is similar to changing a menu item's name.

Study this machine language routine:

```
          PushLong   #NewName      ;Address of title
          PushWord   #2            ;Menu ID number
          _SetMenuTitle            ;Change the title
          _DrawMenuBar             ;show the change
          rts
NewName   str    ' Modem '   ;Pascal-style string
```

The same routine in Pascal:

```
SetMenuTitle(' Modem ', 2);
DrawMenuBar;
```

The same routine in C:

```
SetMenuTitle("\p Modem ", 2);
DrawMenuBar( );
```

SetMenuTitle requires two arguments:

- The address of a Pascal string
- A Menu ID number

Since a Pascal string is needed, the only language that doesn't have to do anything unusual with the string is, of course, Pascal. The machine language example uses the Str macro, while the C example uses the *\p* string escape in order to put a count byte before the new menu title string.

After the title is changed, use DrawMenuBar to show off your handiwork.

Now You See It . . .

Another rarely used feature of the Menu Manager is the ability to insert both menus and menu items into an existing menu structure. This is accomplished with the InsertMenu and InsertMItem functions, respectively. InsertMenu was discussed in detail earlier in this chapter.

To insert a menu item, InsertMItem is used in the following machine language example:

```
          PushLong   #NewItem      ;address of item string
          PushWord   #$FFFF        ;make it the last item
          PushWord   #2            ;ID of the Menu to use
          _InsertMItem             ;insert it
          rts
NewItem   dc     c'--New Item\N281D',i1'0'
```

In Pascal:

```
PROCEDURE InsertNewItem;
VAR
     NewItem : String;
BEGIN
     NewItem := ‘--New Item\N281D\0’;
     InsertMItem(@NewItem[1], $ffff, 2);
END;
```

In C:

```
InsertMItem(“--New Item\N281D”, 0xffff, 2);
```

InsertMItem's three arguments are

- The address to a complete menu item string
- The position where the item should be inserted
- The ID number of the menu to use

Values for the position (second) argument are

Position	**Description**
$0000	Insert into the menu before all other items
$FFFF	Insert into the menu after all other items
ItemID	Insert after the specified Menu Item ID

As described earlier, CalcMenuSize should be called after inserting a new menu item.

. . . Now You Don't

If the Toolbox allows you to insert menus and menu items, there must also be a way to delete them. DeleteMenu and DeleteMItem are practically identical in syntax. They both require a single input parameter:

- A menu ID number for DeleteMenu
- An item ID for DeleteMItem

To delete an entire menu in machine language, use

```
PushWord   #MenuID     ;the ID of the menu to delete
_DeleteMenu            ;it's gone!
```

Using C and Pascal:

```
DeleteMenu(MenuID);
```

To delete a menu item in machine language use

```
PushWord   #ItemID      ;the ID of the item to delete
_DeleteMItem            ;poof!
```

In C and Pascal:

```
DeleteMItem(ItemID);
```

Change Blink Rate

After a menu item is selected, the item winks at you a few times before your choice is acted upon. The number of times the item blinks is determined by the blink rate. Usually, this value is set to 3 upon starting the Menu Manager. But you can spring the following routine on some unsuspecting user.

In machine language:

```
PushWord    #50      ;blink 50 times!
_SetMItemBlink
```

In C and Pascal:

```
SetMItemBlink(50);
```

When SetMItemBlink is used to change the blink rate to 50, a selected menu item will flash on and off 50 times before the item is handled.

Menu Bar Colors

If you're enthralled by the myriad of colors your Apple IIGS can produce, you'll be happy to know that even the menu bar can show its true colors. The text, background, and outlining can be set to any of 16 different colors in 320 mode, and 4 colors in 640 mode. Even though the colors can be changed in 640 mode, it's hardly worth the trouble because so few colors are available. But in 320 mode, the effects can be quite interesting.

How about a blue background, yellow text, and red outlines? Use the MODEL program from Chapter 6 and insert the following code just before the DrawMenuBar function is called.

In machine language:

```
PushWord    #$49     ;Background and text colors
PushWord    #$94     ;Background & text for color replace
PushWord    #$70     ;Outline color
_SetBarColors
```

In C:

```
SetBarColors(0x49, 0x94, 0x70);
```

In Pascal:

```
SetBarColors($49, $94, $70);
```

> To get blue, yellow, and red menu bar colors, the QuickDraw tool set will have to be started up for 320 mode. To do this, use a MasterSCB (screen mode) value of $00. Also, make sure to specify a maximum X clamp of 320 pixels when starting the Event Manager.

SetBarColors uses three input values:

	Colors per Mode		
Value Name	**320**	**640**	**Description**
NewBarColor	16	4	Background (bits 4–7), text (bits 0–3)
NewInvertColor	16	4	Color-replace values for background/text bits
NewOutColor	16	4	Outline color (bits 4–7)

All unused bits are 0, except for bit 15, the most significant bit. This bit is used to cancel the effects of a value. In other words, your program could establish a new outline color, but leave the text and background colors as they were by setting bit 15 on the NewBarColor and NewInvertColor arguments.

When the modified MODEL program runs with new menu colors, the menu bar will be dark blue with yellow text. The outline around the menus, dividing lines, and underlines will be red. But selected menus and items will appear in light blue with orange text.

The Apple menu will retain its colorful logo, but on a yellow background. Why? Recall that the Apple menu uses the special character *X* in its menu string. This denotes a color-replace mode when selected. All other menus and their items use an XOR (eXclusive OR) method of highlighting when selected.

It's clear to see why this occurs by examining Table 8-5. The color number for dark blue is 4. When XORed with its complement (EOR #$FF), the result is 11, which corresponds to light blue. Likewise, yellow (9) XORed with its complement results in orange (6).

Table 8-5. Standard Colors in 320 Mode

Color	Number
Black	0
Dark Gray	1
Brown	2
Purple	3
Dark Blue	4
Dark Green	5
Orange	6
Red	7
Beige	8
Yellow	9
Green	10
Light Blue	11
Lilac	12
Periwinkle	13
Light Gray	14
White	15

The second argument, NewInvertColor, is applicable only to color-replace items. So in order to cause selected items to appear in blue text with a yellow background, the opposite of their backgrounds when not selected, the special *X* character would have to be placed in each item's menu string.

With a little creativity, you could create a menu where only selected items would show up in a color, indicating a warning or other message to the user, based on the color.

There are accepted guidelines governing the use of color in programs. See Appendix A, "Human Interface Guidelines," for more details.

Chapter Summary

The following tool set functions were referenced in this chapter:

Function: $010F
Name: MenuBootInit
Initializes the Menu Manager
Push: nothing
Pull: nothing
Errors: none
Comments: Do not make this call.

Function: $020F
Name: MenuStartUp
Starts the Menu Manager
Push: User ID (W); Direct Page (W)
Pull: nothing
Errors: none

Function: $030F
Name: MenuShutDown
Shuts down the Menu Manager
Push: nothing
Pull: nothing
Errors: none
Comments: Make this call when your application is finished.

Function: $0D0F
Name: InsertMenu
Inserts a menu into the menu bar
Push: Menu Handle (L); Insert After (W)
Pull: nothing
Errors: none
Comments: If Insert After is 0, the menu becomes the first in the menu bar.

Function: $0E0F
Name: DeleteMenu
Removes a menu from the menu list
Push: Menu Number (W)
Pull: nothing
Errors: none
Comments: Use DrawMenuBar to update the screen. The menu is not fully disposed, just deleted from the list.

Function: $0F0F
Name: InsertMItem
Inserts a menu item into a menu
Push: Item (L); Insert After (W); Menu Number (W)
Pull: nothing
Errors: none
Comments: If Insert After is 0, the item becomes the first in the menu.

Function: $100F
Name: DeleteMItem
Removes an item from a menu
Push: Item (W)
Pull: nothing
Errors: none
Comments: Use CalcMenuSize after making this call.

Function: $130F
Name: FixMenuBar
Standardizes the menu bar's sizes and returns its height
Push: Result Space (W)
Pull: Height (W)
Errors: none
Comments: The returned height is in pixels and is usually 13.

Function: $170F
Name: SetBarColors
Specifies the colors of the menu bar
Push: Normal Color (W); Selected Color (W); Outline Color (W)
Pull: nothing
Errors: none

Function: $1C0F
Name: CalcMenuSize
Calculates the new dimensions of a menu
Push: Width (W); Height (W); Menu Number (W)
Pull: nothing
Errors: none

Function: $1F0F
Name: SetMenuFlag
Specifies the attributes of a menu
Push: Attributes (W); Menu Number (W)
Pull: nothing
Errors: none
Comments: Attributes are: $FF7F = Enable, $0080 = Disable; $FFDF = Color Replace; $0020 = XOR Highlight; $FFEF = Standard; $0010 = Custom.

Function: $210F
Name: SetMenuTitle
Selects the title for a menu
Push: Title (L); Menu Number (W)
Pull: nothing
Errors: none

Function: $240F
Name: SetMItem
Selects the name for an item
Push: Name (L); Item Number (W)
Pull: nothing
Errors: none

Function: $260F
Name: SetMItemFlag
Sets the attributes of a menu item such as being underlined, enabled, and so on
Push: Attributes (W); Item Number (W)
Pull: nothing
Errors: none
Comments: Attributes are: $0040 = Underline, $FFBF = No Underline, $0020 = XOR Highlight; $FFDF = Redraw Highlight; $FF7F = Enable; $0080 = Disable.

Function: $280F
Name: SetMItemBlink
Sets the blink rate for selected items
Push: Blink Count (W)
Pull: nothing
Errors: none

Function: $2A0F
Name: DrawMenuBar
Draws the menu bar and its titles
Push: nothing
Pull: nothing
Errors: none

Function: $2C0F
Name: HiliteMenu
Determines if a menu title is highlighted
Push: Hilite Flag (W); Menu Number (W)
Pull: nothing
Errors: none
Comments: If Hilite Flag is nonzero, the title is highlighted; otherwise, it's unhighlighted.

Function: $2D0F
Name: NewMenu
Creates a new menu
Push: Result Space (L); Menu Structure (L)
Pull: nothing
Errors: none
Comments: This creates the menu internally and does not display or insert it into a menu bar.

Function: $300F
Name: EnableMItem
Enables a disabled menu item
Push: Item (W)
Pull: nothing
Errors: none

Function: $310F
Name: DisableMItem
Disables a menu item, making it dimmed
Push: Item (W)
Pull: nothing
Errors: none
Comments: The item will no longer be available for selection.

Function: $320F
Name: CheckMItem
Manages check marks for a menu item
Push: Check Flag (W); Item (W)
Pull: nothing
Errors: none
Comments: An item will be marked with a check if Check Flag is true; a check will be removed if false.

Function: $330F
Name: SetMItemMark
Sets the marking character (or none) for an item
Push: Mark Character (W); Item Number (W)
Pull: nothing
Errors: none
Comments: Use 0 for no mark.

Function: $350F
Name: SetMItemStyle
Sets the text style of a menu item
Push: Text Style (W); Item Number (W)
Pull: nothing
Errors: none

Function: $3A0F
Name: SetMItemName
Selects a name for a menu item
Push: Name (L); Item Number (W)
Pull: nothing
Errors: none
Comments: Name is a Pascal-type string.

Chapter 9

Windows

Next to pull-down menus, windows are the most important part of the desktop environment. A window is a region of the screen inside of which information and/or graphics can be displayed. The Toolbox's Window Manager provides the functions for creating a window and placing various objects into it.

This chapter covers programming, creating, and using Apple IIGS windows. Unfortunately, not everything about windows can be covered here. It would require a gargantuan book to demonstrate everything the Window Manager can do. And, of course, it would take a trilogy of these books to present program examples in three languages as is being done here. You'll find enough routines and examples in this chapter to start experimenting. If you practice, you'll be writing useful window applications of your own in short order.

A Frame to Build On

Windows are controlled by a joint cooperation between the Window Manager and the Control Manager. The Window Manager is what actually manages the windows (as you may have guessed). It also takes care of certain functions that occur behind the scenes. The Control Manager is responsible for all the controls on a window. Controls are the buttons, boxes, scroll bars, and other items that allow you to manipulate a window. Therefore, both managers share the responsibility of windows on the desktop.

To use windows in your Apple IIGS program, you'll need to have already started the Tool Locator, Memory Manager, and Miscellaneous tool sets (the "big three"), as well as QuickDraw II and the Event Manager. After that, you should start the Window Manager and then the Control Manager.

The Window Record

Once all your tool sets have been started, placing a window on the screen isn't a difficult task. In fact, you simply pass information about the window to a Window Manager Toolbox function. The window's information is kept in one of the longest record structures used by the Toolbox, the window record. The window record stores all sorts of information about the window: its size, contents, color, types of controls, movability, ability to zoom, and large quantities of additional information.

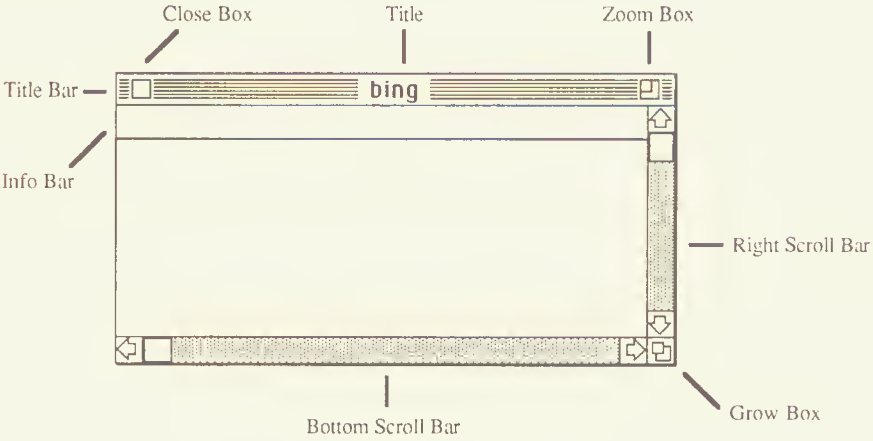
Unlike the Menu Manager, which uses several intervening steps between creating the menu and having it appear on the screen, the NewWindow function displays a window immediately. All you have to do is point to your window record so NewWindow can find it.

NewWindow returns a long pointer to your window's *port* in memory after a successful Toolbox call. Use this pointer to reference the window. For example, to close the window, the port address for that window is pushed onto the stack and a Toolbox call is made to the CloseWindow function. All other Window Manager calls use the port pointer, and there is a port for each window on your desktop.

Things in a Window

When you're creating a window, you should be familiar with all the controls it uses, and with what each control does. These controls are summarized in Figure 9-1.

Figure 9-1. Diagram of Window with Controls



The controls and items inside a window are explained below. All of these items are optional: A window need not contain any of them.

Bottom scroll bar. The bottom scroll bar moves the contents of the window right or left.

Close box. The close box is used to remove the window from the desktop (to make it disappear). This is not a direct function of this control. Your program actually makes the window close. Closing a window is covered in detail later in the chapter. The close box control is located in the title bar.

Grow box. The grow box is used to resize the window. The grow box can be grabbed with the mouse and moved to change the horizontal and vertical dimensions of the window.

Info bar. The info bar appears just below the title bar and is used to display additional information about the window. The Apple IIGS Finder program makes extensive use of window info bars to let you know how many files are present in each window, and so on.

Right scroll bar. The right scroll bar moves, or scrolls, the contents of the window either up or down. Only if a window has contents larger than can be seen through the window does it need a scroll bar.

Title. The title is the name of the window, centered in the title bar.

Title bar. The title bar shows the title of the window. The title bar also contains the optional close box, or *go-away button,* and the zoom button. The title bar is used to drag the window around the desktop. Because of this it's also referred to as the *drag region* of the window.

Zoom box. The zoom box can be used to make the window expand to fill the entire screen. Clicking the zoom a second time restores the window to its previous size. Both sizes, original and zoomed, are determined by the Window Record at the time the window is created.

When you're creating a window, all these items are specified in the window record. Depending on what type of data is in the window and how you want it displayed, any number of these options can be specified.

The TaskMaster

No discussion of windows and controls would be complete without mention of the TaskMaster. TaskMaster is a Window Manager function that acts as an extension of the Event Manager. It's especially handy when you're dealing with windows. Though the TaskMaster is discussed elsewhere in this book, it's important to know the window-related event codes returned by TaskMaster.

The following table shows the extended event codes and regular event codes returned by the TaskMaster function. Note that extended events 2–12 concern themselves with windows.

Table 9-1. Event and Extended Event Codes Returned by TaskMaster

Event Code	Extended Code	Description
16	0	Mouse is in desk
17	1	A Menu item was selected
18	2	Mouse is in the system window
19	3	Mouse is in the content of a window
20	4	Mouse is in drag region
21	5	Mouse is in grow
22	6	Mouse is in go-away
23	7	Mouse is in zoom
24	8	Mouse is in info bar
25	9	Mouse is in vertical scroll
26	10	Mouse is in horizontal scroll
27	11	Mouse is in frame
28	12	Mouse is in drop

When one of these events takes place, the event code is returned by TaskMaster. The window associated with the event can be determined by examining the TaskData field of the event record. For example, if your desktop had many windows on it and you clicked the go-away button in one of them, that window's pointer would be placed in TaskData. The window can be further manipulated by Window Manager functions that use the window pointer. (A good example of this is in the MONDO program listed at the end of this chapter.)

The important thing to remember about TaskMaster is that it assists in the trapping of window-related events. It also automatically updates the contents of a window as you scroll them around.

Opening a Window

Putting a window on the screen is a trivial task. A simple call to the Toolbox is all that is required. The complexity of the window lies in the window record—a group of values, ranges, and pointers that actually define the window.

For example, suppose you wanted to display a typical Apple IIGS window. To do this you need two things:

- A call to the Window Manager's NewWindow function
- The window record describing the window

In machine language, the call looks something like this:

```
pha                             ;Long result space
pha
pushlong       #WindowRec       ;Address of window record
_NewWindow                      ;the new window call
jsr            ErrorH           ;Remember to do error checking
pulllong       WindowPtr        ;A pointer to the window
```

First, a long word of result space is pushed to the stack, followed by the address (long) of the window record. Then the call is made to NewWindow. After the call, the Toolbox returns a pointer to the window's record. All further reference to the window is made through this pointer, so it should be saved in memory. (The above routine saves the pointer at the label *WindowPtr.*)

The only possible errors at this point are $0E01 and $0E02. Error $0E01 is produced if the window record is of an unusual length (meaning you left something out or the pointer was inaccurate). Error $0E02 is a memory error and probably would only happen if your computer didn't have a memory upgrade or if you had too many windows already open.

A typical error in working with structures in machine language, especially if you're using macros, is to reference the address of a structure incorrectly. For example, the following pushlong macro is in error:

```
pushlong    WindowRec    ;This is wrong
```

Because the # in front of *WindowRec* is left off, the program attempts to push the long value that resides at WindowRec. This is akin to leaving off the ampersand (&) before a variable in a C program.

What is intended is that the address of WindowRec (its location in memory) be pushed onto the stack. The address of any object is always referenced as an immediate value. Thus, the following is the correct way to push the long address of a structure or label in memory:

```
pushlong    #WindowRec   ;This is the correct way
```

In C, the following routine can be used to summon up a new window:

```
WindowPtr = NewWindow(&WindowRec);
```

And in Pascal:

```
WindowPtr := NewWindow(WindowRec);
```

As was mentioned earlier, the hard part (if you want to call it that) is creating the window record. It contains a wealth of information about the window and is perhaps the most detailed record used by the Toolbox. The window record is covered later in this chapter.

Closing a Window

All that's needed to close a window is the pointer to the window record and, of course, a call to the Window Manager's CloseWindow function. After CloseWindow is called, the window is removed from the screen and all the data contained in the window is gone. Using the pointers returned from the NewWindow call in the previous section, the following code examples are used to close a window referenced by WindowPtr.

In machine language:

```
pushlong        WindowPtr     ;Saved when the window was opened
_CloseWindow                  ;(No errors are possible here)
```

In C and Pascal:

```
CloseWindow(WindowPtr);
```

After CloseWindow, the window disappears from the screen, it is removed from the current list of windows, and any data contained in the window is lost. A window doesn't have to be on top of all the other windows in the desktop, nor does it have to be active in order to be removed.

It's important to note that clicking in a window's close box does not automatically close the window. Nor does selecting a *close window* option from a pull-down menu. Closing down a window has to be done by the code in your program.

To detect when the close box has been clicked, you must use the Window Manager's TaskMaster function. The extended event codes returned by the TaskMaster call are your clues as to what is going on in a window. Normally, most of the operations (scrolling, growing, moving, zooming, and so on) are taken care of automatically by the operating system. But your program will have to monitor the close box.

Extended event code 6, or regular event code 22, is returned by the TaskMaster call when the mouse is clicked in the close box (see Table 9-1). When extended event code 6 is returned, the corresponding window's pointer is found in the TaskData field of the event record. To close the window, the following machine language code can be used:

```
pushlong        TaskData    ;get window's pointer from TaskData.
_CloseWindow
```

The window record, placed in TaskData by the TaskMaster, is pushed to the stack for the CloseWindow call. This is the same as a regular close, except the window pointer is snatched from TaskData.

As usual, the examples for C and Pascal are a little more straightforward.

In C:

```
CloseWindow(EventRec.wmTaskData);
```

In Pascal:

```
CloseWindow(WindowPtr(EventRec.TaskData));
```

This method of using TaskData works even when there are a number of windows present on the desktop.

The Window Record

The window record defines the window, determines what the window can do, and establishes which controls (zoom, grow box, title bar, and so on) the window will have. The window record is huge—24 parameters determine what type of window is created.

In the following table, the parameter name is the word used by Apple in all documentation to refer to that particular parameter of the window record. Later on, when a sample window record is created, a few of the parameters will be combined into one to make the list easier to manage.

Table 9-2. The Window Record's Parameter List

Parameter Name	Type	Description
paramlength	Word	Size of this table
wFrame	Word	Bit pattern describing the frame
wTitle	Long	Window's title
wRefCon	Long	User-defined value, usually 0
wZoom	Rectangle	Size of window when zoomed
wColor	Long	Window's color table location
wYOrigin	Point	Window content's origin, Y position
wXOrigin	Point	Window content's origin, X position
wDataH	Word	Height of document
wDataW	Word	Width of document
wMaxH	Word	Maximum height for grow window
wMaxW	Word	Maximum width for grow window
wScrollV	Word	Number of Y pixels to scroll
wScrollH	Word	Number of X pixels to scroll
wPageVer	Word	Number of Y pixels to page
wPageHor	Word	Number of X pixels to page
wInfoRefCon	Long	Used by info-bar draw routine
wInfoHeight	Word	Height of info-bar
wFrameDefProc	Long	Window definition procedure
wInfoDefProc	Long	Info-bar drawing routine
wContDefProc	Long	Content drawing procedure
wPosition	Rectangle	Window's starting coordinates
wPlane	Long	Position, front to back
wStorage	Long	Memory for window record

Incidentally, the tiny *w* at the front of a parameter name is an instant tip-off that the parameter belongs to the Window Manager.

Each of the parameters is discussed in detail in *COMPUTE!'s Mastering the Apple IIGS Toolbox*. However, the following is a brief rundown of each of them, along with explanations and expansions where necessary.

paramlength. The parameter paramlength (word value) is the length of the entire window record. It's used by the Memory Manager in moving these parameters to the internal window record. It also serves as a form of error checking: If the paramlength is inaccurate, the Window Manager returns an error code of $0E01 after the NewWindow call.

wFrame. The parameter wFrame (word value) describes the frame of the window. Each bit in the word wFrame signals the presence or absence of one of the window controls. A window with everything on it has the following bit pattern:

```
1101111110100000
```

which is $DFA0 in hex. The individual significance of each of the bits is shown in Table 9-3.

Table 9-3. wFrame Values

Bit	If set, means
0	The window is highlighted (initially always 0)
1	Window is zoomed when first drawn
2	Internal use (determines window record allocation)
3	Window's controls can be active when the window is inactive
4	The window has an info bar
5	The window is visible
6	An inactive window is made active if the mouse is clicked in it
7	The window can be moved (bit 15 should also be set)
8	The window has a zoom box (bit 15 should also be set)
9	The size of the window is flexible (grow and zoom will not change the origin of the window's data)
10	The window has a grow box (bit 11, bit 12, or both should also be set)
11	The window has an up- and down-scroll bar (right side)
12	The window has a left- and right-scroll bar (bottom)
13	The window has a double frame, like an alert dialog box
14	The window has a go-away button (bit 15 should also be set)
15	The window has a title bar

The bits above that are set to define a window "with the works" are 5, 7, 8, 9, 10, 11, 12, 14, and 15. Bit 13 is used by the Dialog Manager when it creates a window. Bits 4, 8, 9, 10, 11, 12, 14, and 15 must be reset to 0 if this bit is set.

wTitle. wTitle (long pointer) points to the memory location containing the window's title. The title is a Pascal string, and it's a good idea to pad it with spaces. (This keeps the title from appearing too tight in the title bar. More on this in a while.) If a long word of 0 is specified, the window has no title.

wRefCon. wRefCon (long value) is a user-defined value, though typically a long word of 0 is specified. A few of the Window Manager's functions can return or set this value, but its meaning is up to you. For example, in the sample program, MONDO, wRefCon is used to number each window for later reference in the program.

wZoom. wZoom (rectangle) indicates the size of the window when zoomed. The four word values are listed in the order MinY, MinX, MaxY, MaxX. If four words of 0 are specified, the entire screen is filled with the window. It's also suggested your window have a zoom box (bit 8 of wFrame above).

wColor. wColor (long pointer) points to a table controlling the color of the window, title bar, and frame. If a long word of 0 is specified, the system default colors are used. (See the section on color later in this chapter for additional information.)

wYOrigin and wXOrigin. wYOrigin (point) and wXOrigin (point) set the Y and X origins of the window's data. Both Y and X are word values, expressed in global coordinates (with 0, 0 as the upper left corner of the screen). In this book, both wYOrigin and wXOrigin together are referred to as the point value wOrigin.

wDataH and wDataW. wDataH (word) and wDataW (word) designate the height and width of the data inside the window. If the data cannot be scrolled (meaning the window doesn't have scroll bars), two words of 0 are used. wMaxH (word) and wMaxW (word) specify the maximum height and width of the window. The size of the window is manipulated by the window's grow box and is measured in pixels.

wScrollV and wScrollH. wScrollV (word) and wScrollH (word) define the number of Y and X pixels, respectively, that a window may scroll when the arrows are clicked in either the up/down or left/right scroll bars.

wPageVer and wPageHor. wPageVer (word) and wPageHor (word) define the number of Y and X pixels that a window is paged. Paging occurs when the mouse is clicked inside a scroll bar. (This should be a proportionally larger value than for wScrollV and wScrollH, above.)

wInfoRefCon. wInfoRefCon (long pointer) points to a string to be placed in the window's information bar. If there is no string, it points to a long word of 0. (The window should have an information bar for this value to take effect—bit 4 of wFrame above.)

wInfoHeight. wInfoHeight (word) defines the height, in pixels, of the window's information bar. As with wInfoRefCon, the window should contain an information bar for this value to have any meaning.

wFrameDefProc. wFrameDefProc (long pointer) points to a window definition routine or procedure. It is normally set to a long word of 0 to use the default routines.

wInfoDefProc. wInfoDefProc (long pointer) points to a routine that draws the window's information bar, or it points to a long word of 0 if no info bar is present in the window.

wContDefProc. wContDefProc (long pointer) contains the address of a routine that draws the contents of a window. An example of such a routine is listed in the section "Window Contents" later in this chapter. If no routine is used, a long word of 0 is specified. Also, if you don't supply a redraw routine, your window shouldn't have scroll bars.

wPosition. wPosition (rectangle) defines the starting position and size of the window. The four word values are listed in the order MinY, MinX, MaxY, MaxX, and are in global coordinates.

wPlane. wPlane (long value) indicates this window's precedence—in other words, how many windows are stacked on top of it. A long word of 0 places the new window behind every other window on the desktop. A long word of $FFFFFFFF, which is also −1, places the new window on top of all the others.

wStorage. wStorage (long pointer) represents the address of additional storage for the window record. This value is always set to 0 because Apple has not officially said what other values will mean in the future.

The following are examples of window records. Each corresponds to the NewWindow toolbox calls demonstrated earlier in this chapter. To add a window to your program, simply add the NewWindow call as shown above and have it reference a window record with the data you desire. The following window records are simple, standard window records. Later in this chapter, more exciting, splashy, and mind-boggling window records are used.

In machine language:

```
WindowRec   anop
            dc    i'WRecEnd—WindowRec'      ;size of parameter list
            dc    i'%1101111110100000'      ;frame type
            dc    i4'Wtitle'                ;Title string pointer
            dc    i4'0'                     ;Reserved
            dc    i2'0,0,0,0'               ;Position When Zoomed (0=def)
            dc    i4'0'                     ;Pointer to color table
            dc    i2'0,0'                   ;Contents Vert/Horz Origin
            dc    i2'200,640'               ;Height/Width of document
            dc    i2'200,640'               ;Height width for grow window
            dc    i2'4,16'                  ;Vert/horz pixels for scroll
            dc    i2'40,160'                ;vert/horz pixels scroll page
            dc    i4'0'                     ;Value passed to information
            dc    i2'0'                     ;Height of info bar
            dc    i4'0'                     ;Window Definition
            dc    i4'0'                     ;Draw info bar routine
            dc    i4'0'                     ;Draw Interior
            dc    i2'40,100,159,540'        ;Starting position and size
            dc    i4'$FFFFFFFF'             ;Starting plane
            dc    i4'0'                     ;window record
WRecEnd     anop
```

A title string (called Wtitle in the program example) also needs to be defined. The title string is a Pascal string, which means it's preceded by a count byte. Below, the macro *str* is used to define the title for this window:

```
Wtitle    str" Mr. Mondo "
```

Note that the title is padded with spaces. If the spaces were removed, the title would appear jammed into the title bar.

In C, global record structure can be used to create a window record as follows:

```
ParamList   WindowRec = {
            sizeof(WindowRec),      /* size of parameter list */
            0xdfa0,                 /* frame type */
            "\p Mr. Mondo ",        /* Pascal title string */
            NULL,                   /* refcon */
            0, 0, 0, 0,             /* Position When Zoomed (0=def) */
            NULL,                   /* Pointer to color table */
            0, 0,                   /* Contents Vert/Horz Origin */
            200, 640,               /* Height/Width of document */
            200, 640,               /* height/width for grow window */
            4, 16,                  /* vert/horz pixels for scroll */
            40, 160,                /* vert/horz pixels scroll page */
            NULL,                   /* information bar string */
            0,                      /* Height of info bar */
            NULL,                   /* Window Definition routine */
            NULL,                   /* Draw info bar routine */
            NULL,                   /* Draw content routine */
            40, 100, 159, 540,      /* Starting position and size */
            -1L,                    /* starting plane */
            NULL                    /* window record address */
};
```

In Pascal, your window record and its title string must first be declared in the VAR section of your program:

```
VAR
    WindowRec:  NewWindowParamBlk;
    TheTitle:   String;
```

The structure is then loaded with data at runtime (within a function or procedure) with the desired values:

```
TheTitle := ' Mr. Mondo ';

WITH WindowRec DO BEGIN
  param_length   := sizeof(NewWindowParamBlk);
  wFrame         := $dfa0;           { frame type }
  wTitle         := @TheTitle;       { pointer to title string }
  wRefCon        := nil;             { refcon }
  SetRect        (wZoom, 0, 0, 0, 0);
  wColor         := nil;             { color table pointer }
  wYOrigin       := 0;               { content origin }
  wXOrigin       := 0;
  wDataH         := 200;             { document size }
  wDataW         := 640;
  wMaxH          := 200;             { grow window size }
  wMaxW          := 640;
  wScrollVer     := 4;               { scroll range }
  wScrollHor     := 16;
  wPageVer       := 40;              { page range }
  wPageHor       := 160;
  wInfoRefCon    := LongInt(nil);    { Draw info bar routine }
  wInfoHeight    := 0;               { Height of info bar }
  wFrameDefProc  := nil;             { Window Definition routine }
  wInfoDefProc   := nil;             { Draw info bar routine }
  wContDefProc   := nil;             { Draw content routine }
```

```
   SetRect      (wPosition, 100, 40, 540, 159);
   wPlane       := -1;               { starting plane }
   wStorage     := nil;              { window record address }
END;
```

Once you've defined an acceptable window record, you can use it as a template for any other window applications you write. Customizing an existing window record is easier than building a new one each time from the ground up. The remaining sections in this chapter augment certain parameters of the window record, and help make your windows more exciting.

Naming a Window

About the only important thing to do when naming a window is to place spaces on either side of the title. The spaces provide adequate breathing room between the title and the rest of the title bar.

The title of the window is specified in the window record, in the wTitle field:

```
wTitle   (long pointer)
```

wTitle points to the address of a Pascal string that contains the window's title. In the previous window record example, the title of the window was listed as follows (in machine language):

```
   dc   i4'wTitle'   ;title string pointer
```

The actual title is at the address wTitle:

```
wTitle   str' My Window '
```

The *str* macro is used to create a Pascal string with a leading count byte for *My Window.*

You can name a window anything. Or, if you use a long word of 0 for the wTitle field of the window record, the window won't have a title. (This also holds true if you haven't specified a title bar for the window.)

Most often, you'll want to use the name of the file you're working on as the title of the window. This involves a little bytewise sleight of hand in machine language because of the way ProDOS stores a filename in memory. C and Pascal programmers can use standard string-handling functions which make this job a cinch.

A ProDOS filename is stored as a Pascal string, and can be from 1 to 15 characters long (not including a prefix). The characters after the count byte make up the actual name of the file. Since a filename buffer can take up as many as 16 characters (the count byte plus the 15 letters), your programs should provide at least that much space for that worst-case scenario. The string buffer should always be 16 characters long no matter how long the expected filename is.

Suppose you've used the Standard Files tool set to return the name of a file on disk—either a text, picture, or some other file. When the file's name is returned, it is a maximum of 16 characters long, the first of which is a count byte. The filename can be as many as 15 characters long; if there are fewer characters, the remaining characters are padded with nulls (zero bytes).

The object for you, the programmer, is to examine the filename string returned by the Standard Files tool set and make it suitable as the title of a window. Of course, you can't just use the filename returned by ProDOS. Instead, you must delicately extract the filename, being careful to add a space in front and a space behind for padding. And, don't forget to add 2 to the count byte. But the hard part is done for you, as explained below.

In machine language, the following routine moves a ProDOS filename from its storage buffer to a window title storage buffer (the ProDOS filename is stored at location Fname; the window's title, at location wTitle):

```
Pro2Win   LONGA    OFF          ;ProDOS-to-Window Title
          LONGI    OFF
          sep      $30          ;use eight-bit registers
          lda      Fname        ;get the name's length in A
          tax                   ;save a copy in X
          inc      A            ;increase the length by 2
          inc      A
          sta      wTitle       ;save it in the window's title
          lda      $20          ;add a space
          sta      wTitle+1     ;to the beginning of the title
          sta      wTitle+2,x   ;and one to the end
```

```
loop        lda       Fname,x        ;read a character from filename
            sta       wTitle+1,x     ;store filename in title buffer
            dex                      ;work backwards to start of name
            bne       loop           ;if not zero, keep looping

            rep       $30            ;back to 16-bit registers
            LONGA     ON
            LONGI     ON
            rts                      ;and you're done
```

This routine takes the filename from location Fname and moves it to the window's title location, wTitle, and adds one space on each end of the filename. The size of the wTitle buffer should be 18 characters, two more than the Fname buffer, so that it can hold the largest possible filename.

First the routine reads the length of the filename; then it adds 2 to that length (with two INC A instructions), one for each space. Then, before the file's name is transferred, a space is placed at the start and end of the window title.

In the main program loop, the characters are moved from Fname to wTitle. Each character is taken from the right side of Fname, indexed by X, and it works to the left. When X reaches 0, the last character has been moved. This backwards copying method saves a few instructions that would have been needed if you were copying in a forward direction.

In C, the logic follows that of machine language since C's string-handling functions aren't meant to be used with Pascal strings. The routine is as follows:

```
Pro2Win()
{
        char x = Fname[0];                 /* x = length of filename */
        wTitle[0] = x + 2;                 /* set wTitle's new length */
        wTitle[1] = wTitle[x + 2] = ' ';   /* pad with spaces */
        for (; x; --x)                     /* while x is not zero, */
          wTitle[x + 1] = Fname[x];        /* copy the string */
}
```

Far simpler, because of the compatible string format, the following single statement can be used in Pascal:

```
wTitle := CONCAT(' ', Fname, ' ');
```

Feel free to include these routines in any of your programs that use a filename as the window's title.

Colorful Windows

Windows on the Apple IIGS come in several styles, from the plain, black-titled windows, to stylish and colorful windows that would make a Macintosh owner green with envy.

The color of the window is set by the wColor parameter found in the window record:

```
wColor    (long pointer)
```

wColor points to the address of a table containing the colors to be used in the window. If a long word of 0 is specified, the Window Manager creates a black and white window with a solid black title bar.

But you can change that. For example, the following could be included as the wColor parameter of a window record:

```
    dc   i4'WColor'   ;address of color table
```

The color table consists of five word-sized values, each of which describes a different color attribute of the window. The actual colors are determined by the bit positions within each of the words. The five words are described in Table 9-4.

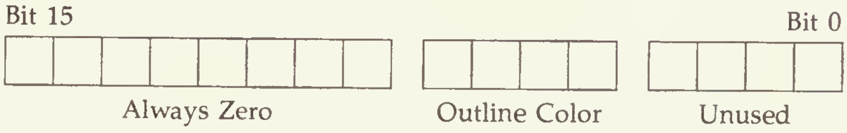
Table 9-4. Color Table

Color Word	Sets the Color For
FrameColor	Window outline
TitleColor	Title, zoom, close boxes
TBarColor	Title pattern and background
GrowColor	Grow box
InfoColor	Info bar

The bit positions are significant in each of these words. Generally speaking, each word is split into groups of four bits (one nibble). These four bits can represent 16 values from 0 (0000) – 15 (1111). Each value then represents one color from the current palette as set by QuickDraw II.

FrameColor (Figure 9-2) sets the color of the window's outline, including the outline of the title bar and info bar.

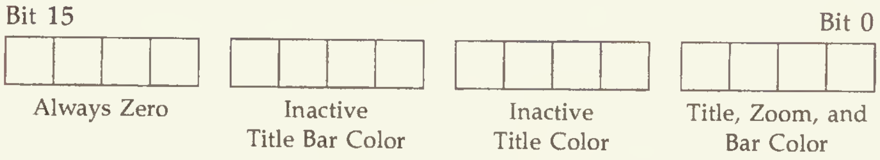
Figure 9-2. Meaning of Color Bits in FrameColor


Bit 15
Bit 0
Always Zero
Outline Color
Unused

The only bits of any significance here are at positions 4–7. Those values control the color of the window's frame. The other bits in this word should be set to 0.

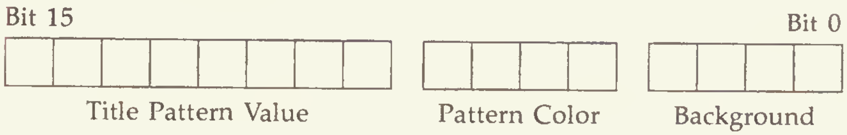
TitleColor (Figure 9-3) controls the color of the window's title (the name of the window), and the close and zoom buttons, as well as the colors of the title bar and title when the window is inactive.

Figure 9-3. Meaning of Color Bits in TitleColor


Bit 15
Bit 0
Always Zero
Inactive Title Bar Color
Inactive Title Color
Title, Zoom, and Bar Color

The inactive colors specified by TitleColor only appear when a window is inactive, or in the background. Otherwise, only bits 0–3 are used when a window is first displayed to color the title, zoom, and close boxes. The inactive title bar color and inactive title color are best used when both values are opposites, such as 0000 for the first and 1111 for the second. When both are the same, the title of an inactive window appears all one color.

Figure 9-4. Meaning of Color Bits in TBarColor

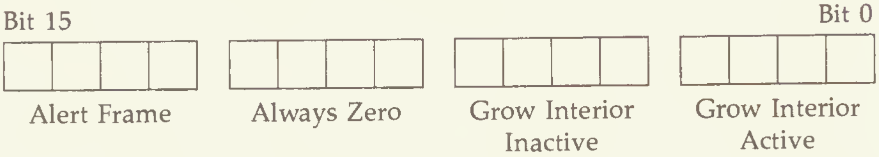

Bit 15
Bit 0
Title Pattern Value
Pattern Color
Background

In the TBarColor slot (Figure 9-4), the title pattern value is one of three values: 0 (00000000) for solid, 1 (00000001) for dithered, or 2 (00000010) for barred—as on the Macintosh.

The pattern color and background (Figure 9-4) set the foreground and background colors for whatever type of pattern is selected. For the solid title bar, only Pattern color (bits 4–7) are used. For a dithered or barred pattern, both values are used.

In the GrowColor Slot (Figure 9-5), the alert frame, bits 12–15, are used when the type of window created is a dialog box and not an actual window. (Remember, the Window Manager is also responsible for creating dialog boxes.)

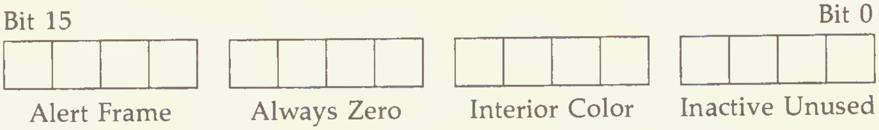
Figure 9-5. Meaning of Color Bits in GrowColor



A special type of dialog box, the alert box, has a few outlines. The alert frame parameter above colors the alert box's middle outline. The grow interior inactive parameter colors the inactive window's grow box. The grow interior active parameter colors the active window's grow box.

As with GrowColor, bits 12–15 of InfoColor (Figure 9-6) determine the color of an alert box. This time the parameter affects the inside outline's color.

Figure 9-6. Meaning of Color Bits in InfoColor



The only other significant bits are 4–7, which control the interior color of the window when it's inactive.

A sample color table would be as follows. In machine language, the percent sign is used to indicate a bit value description of the word. The following color table creates a typical Macintosh-style window using only black and white color values.

```
WColor  dc  i'%0000000000000000'  ;frame color
        dc  i'%0000111100000000'  ;Title Color
        dc  i'%0000001000001111'  ;Title Bar Color
        dc  i'%0000000011110000'  ;Grow Box Color
        dc  i'%0000000011110000'  ;Info Bar Color
```

In C, a sample color table declaration would be

```
WindColor  wColor = {
           0x0000,     /* frame color */
           0x0f00,     /* title color */
           0x020f,     /* title bar color */
           0x00f0,     /* grow box color */
           0x00f0 };   /* info bar color */
```

And in Pascal (wColor is defined as a WindowColorTbl type):

```
WITH wColor DO BEGIN
  FrameColor  := $0000;
  TitleColor  := $0F00;
  TBarColor   := $020F;
  GrowColor   := $00F0;
  InfoColor   := $00F0;
END;
```

Using these premanufactured structures, you'll find it easy to experiment until you create the right window for your needs.

Window Contents

What good is a window unless you can put something into it? Not much. Putting data into a window isn't that difficult; it just requires that you know which buttons to push.

The contents of a window are drawn by a routine indicated in the window record. The wContDefProc parameter contains the long address of a routine that draws the window's contents:

```
wContDefProc   (long pointer)
```

The routine, if written in machine language, should end in an RTL instruction. Functions and procedures in C and Pascal always end in RTLs. The wContDefProc routine draws the contents of the window, then exits. There are no input or output parameters, nor do you need to do any extensive graphics tweaking.

The wContDefProc routine is called by the TaskMaster to update the window's contents—for example, when the window is scrolled or its size is changed. When you're using graphics, the window's port will be the current GrafPort.

If wContDefProc is a long word of 0, then the window will be blank, and scrolling about in the window will erase the window's data. This is why windows without a wContDefProc routine should not have scroll bars.

The following are two examples of the wContDefProc parameter in a window record. The first is an empty field, meaning no procedure is defined. The second is the address of a procedure to draw the contents of a window.

In machine language:

```
dc   i4'0'           ;no update routine
```

or

```
dc   i4'WContent'   ;address of update routine
```

In C:

```
WindowRec.wContDefProc = NULL;          /* no update routine */
```

or

```
WindowRec.wContDefProc = WContent;     /* name of function */
```

In Pascal:

```
WindowRec.wContDefProc := nil;           { no update routine }
```

or

```
WindowRec.wContDefProc := @WContent;  { address of procedure }
```

See the MONDO program example in the next section for a wContDefProc routine that displays a string in a window. When designing your own update routines, remember that the actual size of the window's port is set by the window record when the window is created.

Also, for some reason, having a wContDefProc routine that is just an RTL instruction (or a null routine or procedure in C or Pascal) doesn't seem to work. It causes the machine to crash. Apparently some window access or graphics interaction is required by the routine. This could be because of the versions of tool sets that the IIGS uses at the time of this writing.

The MONDO Window Program

Below is the program MONDO as written in machine language, C, and Pascal. It uses the program MODEL (introduced earlier in this book) as a base upon which to work. To run the MONDO program, you'll need to copy and rename the MODEL program and merge in the following modifications. The program is only altered a little, so there need not be much retyping.

MONDO adds two windows to the MODEL program and performs some interesting trickery with pull-down menus. The windowing routines are used to open and close two separate windows. An extra routine has been added to hide or show the first window and to demonstrate how easy the Window Manager's functions are to use.

When you begin experimenting with this program, change the Show/Hide menu option to some other Window Manager function (SelectWindow, BringToFront, SendBehind, HiliteWindow, or a number of others.

MONDO also demonstrates some interesting Menu Manager functions. For example, when a window is open, its corresponding open menu item is dimmed. When the window is closed, its close menu item is dimmed. This is done with two Menu Manager calls, EnableMItem and DisableMItem. Additionally, when the first window is hidden, the Hide menu item changes to Show. This is accomplished with the SetMItemName function and can be seen in the code samples below.

Another interesting thing to note is how the first window makes use of a custom color table and the second window uses a default color table. Also, note how the wRefCon value is used to identify each window. Because wRefCon's value can be anything you want, MONDO uses it to identify which window is being closed in order to update (dim or enable) the associated menu item. This is done with the GetWRefCon function in the CloseW procedure in the following source code listings.

As usual, feel free to modify this program or use its routines in your own applications. As a special project, try to fix the error that occurs when an invisible window is closed and the Show menu item is not changed back to Hide.

Program 9-1. Machine Language Source for MONDO.ASM

```
*-------------------------------------------------------*
*                      MONDO.ASM                        *
*   Sample Desktop Application in APW Assembler (1.0)   *
*                  Windowing Routines                   *
*-------------------------------------------------------*

: To create the Mondo.Macros macro file, use this APW shell command:
; # macgen mondo.asm mondo.macros 2/ainclude/m16=

        ABSADDR ON
        KEEP    Mondo
        MCOPY   Mondo.Macros

*------------------------------*
*       Global Equates         *
*------------------------------*

Toolbox gequ    $e10000         ;Primary tool dispatcher
TRUE    gequ    $8000           ;True value
FALSE   gequ    $0000           ;False value
Page    gequ    $100            ;The size of a page (256 bytes)
mOpen1  gequ    257
mHide1  gequ    258
mClose1 gequ    259
mOpen2  gequ    260
mClose2 gequ    261

; *NOTE*     From this point on, copy the source from
; *NOTE*     the original MODEL.ASM APW program...

ModelA  START
        phk                     ;Make the data bank...
        plb                     ;...the current code bank
        brl     Main            ;branch over functions to Main

; *NOTE*     Et cetera, on down to the end of the "Main" routine
; *NOTE*     as follows:

        jsr     ShutDownTools   ;Shut down all tools started

        _QUIT   Qparms          ;Exit this program through ProDOS 16

; *NOTE*     Then add what follows after this point.
; *NOTE*     It augments and replaces the rest of the ModelA
; *NOTE*     source code:

*------------------------------*
*      Window Routines         *
*------------------------------*

Open1   pea $0000               ;long result space
        pea $0000
        pushlong #WindRec1      ;first window record
        _NewWindow              ;open it

        jsr ErrChk              ;check for errors
        pulllong WindPtr1       ;get pointer #1
        pea mOpen1              ;this menu item number
        _DisableMItem           ;dim it

        pea mHide1              ;enable these two
        _EnableMItem

        pea mClose1
        _EnableMItem
        rts

;------------------------------

hidebit dc i2'0'                ;zero = window is visible

Hide1   lda hidebit
        beq HideIt

ShowIt  pushlong WindPtr1       ;show the window
        _ShowWindow

        pushlong #mHide         ;change the name back
        pea mHide1
        _SetMItemName

        lda #$0
        sta hidebit
        rts

HideIt  pushlong WindPtr1       ;hide this window
        _HideWindow             ;hide it, no errors

        pushlong #mShow         ;change menu item name
        pea mHide1              ;this item number
        _SetMItemName           ;change the name

        lda #$FFFF              ;change status byte
        sta hidebit
        rts

;------------------------------

Open2   pea $0000               ;long result space
        pea $0000
        pushlong #WindRec2      ;second window record
        _NewWindow              ;open that window
        jsr ErrChk
        pulllong WindPtr2       ;get second window's pointer

        pea mOpen2              ;now, dim this menu
        _DisableMItem

        pea mClose2             ;and enable this item
        _EnableMItem
        rts

;------------------------------

Close1  pushlong WindPtr1       ;close this window
        _CloseWindow
        pea mOpen1              ;this menu item number
        _EnableMItem            ;Enable this again

        pea mHide1              ;disable these two
        _DisableMItem

        pea mClose1
        _DisableMItem
        rts

;-------------------------------

Close2  pushlong WindPtr2       ;close window two
        _CloseWindow

        pea mOpen2              ;re-enable this item
        _EnableMItem

        pea mClose2
        _DisableMItem
        rts

;-------------------------------
; close whichever window was clicked

CloseW  pea $0000               ;long result space
        pea $0000
        pushlong TaskData       ;get the window's pointer from Taskdata..
        _GetWRefCon             ;get the window's wRefCon value

        pla                     ;get low order word into a
        plx                     ;toss away high order word in x
        cmp #1                  ;if 1, then it's window 1
        beq Close1              ;so close window 1
        bra Close2              ;else, close window 2

;-------------------------------
WContent pea $0028                      ;Horizontal pointer loc.
         pea $0020                      ;Vertical pointer loc.
         _MoveTo                        ; (QuickDraw)

         PushLong #String0
         _DrawCString

         rtl                            ;long return

String0  dc c"This is a string inside the window.",i1'0'

*-------------------------------*
*      Variable Storage         *
*-------------------------------*

UserID  ds      2               ;Our User ID
MemID   ds      2               ;Memory User ID (made from User ID)
DPBase  ds      2               ;Used by DP buffer manager
QFlag   dc      i'FALSE'        ;Boolean: Quit flag (starts out as false)

*===============================*
*  StartUp/Shutdown Tool List   *
*-------------------------------*
```

```
Toolist dc      i'(ToolstE-Toolist-1)/4'        ;Tool count
        dc      i'1,0'          ; Tool Locator
        dc      i'2,0'          ; Memory Manager
        dc      i'3,0'          ; Misc Tools
        dc      i'4,0'          ; QuickDraw II
        dc      i'6,0'          ; Event Manager
        dc      i'14,0'         ; Window Manager
        dc      i'16,0'         ; Control Manager
        dc      i'15,0'         ; Menu Manager
        dc      i'5,0'          ; Desk Manager
ToolstE         anop

*------------------------------*
*   Pull Down Menu Structures  *
*------------------------------*

MenuTbl dc      i'(MenTblE-MenuTbl-1)/2'        ;Menu count
        dc      i'Menu1'        ;Apple
        dc      i'Menu2'        ;Window
        dc      i'Menu3'        ;Quit
MenTblE anop

Menu1   dc      c'>>@\XN1',i1'0'                ;Apple
        dc      c'--About This Program...\N256',i1'0'
        dc      c'---\D',i1'0'
        dc      c'>'

Menu2   dc      c'>>  Window  \N2',i1'0'        ;Window
        dc      c'--Open Window1\N257',i1'0'
        dc      c'--Hide Window1\DN258',i1'0'
        dc      c'--Close Window1\DVN259',i1'0'
        dc      c'--Open Window2\N260',i1'0'
        dc      c'--Close Window2\DN261',i1'0'
        dc      c'>'

Menu3   dc      c'>>  Quit  \N3',i1'0'          ;Quit
        dc      c'--Quit\N262*Qq',i1'0'
        dc      c'>'

mShow   str 'Show Window1'                      ;changing menu items
mHide   str 'Hide Window1'

*------------------------------*
* Menu Item Dispatch Addresses *
*------------------------------*

MTable  dc      i'About'        ;256/About              (Apple Menu)
        dc      i'Open1'        ;257/Open window 1      (Window Menu)
        dc      i'Hide1'        ;258/Hide window 1
        dc      i'Close1'       ;259/Close window 1
        dc      i'Open2'        ;260/Open window 2
        dc      i'Close2'       ;261/Close window 2
        dc      i'Quit'         ;262/Quit               (File Menu)

*------------------------------*
*       The Event Record       *
*------------------------------*

EventRec        anop                    ;Event Record used by TaskMaster
EWhat           ds      2               ;What
```

```
EMsg            ds      4                   ;Message
EWhen           ds      4                   ;When
EWhere          ds      4                   ;Where
EMods           ds      2                   ;Modifiers
TaskData        ds      4                   ;Task Data
TaskMask        dc      i4'$1fff'           ;Task Mask

*-----------------------------*
*       Window Data           *
*-----------------------------*

WindPtr1 ds 4                               ;Window pointer

W1title  str " Mr. Mondo One "

W1Ctable dc i'%0000000000000000'            ;frame color
         dc i'%0000111100000000'            ;Title Color
         dc i'%0000001000001111'            ;Title Bar Color
         dc i'%0000000011110000'            ;Grow Box Color
         dc i'%0000000011110000'            ;Info Bar Color
WindRec1 anop
         dc i'WR1end-WindRec1'              ;size of parameter list
         dc i'%1101111110100000'            ;frame type
         dc i4'W1title'                     ;title
         dc i4'1'                           ;Used for window number here
         dc i2'0,0,0,0'                     ;Position When Zoomed 0=def
         dc i4'W1Ctable'                    ;Pointer to color table
         dc i2'0,0'                         ;Contents Vert/Horz Origin
         dc i2'180,640'                     ;Height/Width of document
         dc i2'180,640'                     ;height/width for grow window
         dc i2'4,16'                        ;vert/horz pixels for scroll
         dc i2'40,160'                      ;vert/horz pixels scroll page
         dc i4'0'                           ;Value passed to information draw
         dc i2'0'                           ;Height of info bar
         dc i4'0'                           ;Window Definition
         dc i4'0'                           ;Draw info bar routine
         dc i4'WContent'                    ;Draw Interior
         dc i'40,100,159,540'               ;Starting position and size
         dc i4'$FFFFFFFF'                   ;starting plane
         dc i4'0'                           ;Window Record
 WR1end  anop

 WindPtr2 ds 4                              ;Window pointer

 W2title  str " Mr. Mondo Two "

 WindRec2 anop
          dc i'WR2end-WindRec2'             ;size of parameter list
          dc i'%1101111110100000'           ;frame type
          dc i4'W2title'                    ;title
          dc i4'2'                          ;Used for window number here
          dc i2'0,0,0,0'                    ;Position When Zoomed 0=def
          dc i4'0'                          ;no color table
          dc i2'0,0'                        ;Contents Vert/Horz Origin
          dc i2'180,640'                    ;Height/Width of document
          dc i2'180,640'                    ;height/width for grow window
          dc i2'4,16'                       ;vert/horz pixels for scroll
          dc i2'40,160'                     ;vert/horz pixels scroll page
          dc i4'0'                          ;Value passed to information draw
          dc i2'0'                          ;Height of info bar
          dc i4'0'                          ;Window Definition
         dc i4'0'                        ;Draw info bar routine
         dc i4'0'                        ;Draw Interior (none)
         dc i'50,120,169,560'            ;Starting position and size
         dc i4'$FFFFFFFF'                ;starting plane
         dc i4'0'                        ;Window Record
WR2end   anop

*------------------------------*
*      Miscellaneous Data      *
*------------------------------*

Moment  dc      c'One Moment...',i1'0'

QParms  dc      i4'0'          ;ProDOS 16 Quit Code parameters
        dc      i'$0000'

        END
```

Program 9-2. C Language Source for MONDO.C

```
/*---------------------------------------------*
 *                  MONDO.C                    *
 * Sample Desktop Application in APW C (1.0)   *
 *---------------------------------------------*/

/* **NOTE** This is not a complete program.  Merge parts of
            this listing with the MODEL.C program from
            Chapter Six.  Insert portions from MODEL.C where
            indicated. */

/* #include directives -- insert from MODEL.C */

#define mAbout  256         /* Menu item IDs */
#define mOpen1  257
#define mHide1  258
#define mClose1 259
#define mOpen2  260
#define mClose2 261
#define mQuit   262

/*------------------------------*
 *      Global Variables        *
 *------------------------------*/

WmTaskRec    EventRec;            /* Event Record Structure */

Word         Event,               /* Event code */
             UserID,              /* Our User ID */
             MemID,               /* Memory Management ID */
             QFlag;               /* Boolean: Quit flag */

Word         Toolist[] = (
                 3,               /* Tool count */
                 14, 0,           /* Window Manager */
                 15, 0,           /* Menu Manager */
                 16, 0            /* Control Manager */
);

char         *DPBase;             /* Direct Page base pointer */
```

```
GrafPortPtr WindPtr1,           /* Window port pointers */
            WindPtr2;

WindColor   W1Ctable = {
                0x0000,         /* frame color */
                0x0f00,         /* title color */
                0x020f,         /* title bar color */
                0x00f0,         /* grow box color */
                0x00f0          /* info bar color */
};

int WContent();
ParamList   WindRec1 = {
                sizeof(WindRec1),   /* size of parameter list */
                0xdfa0,             /* frame type */
                "\p Mr. Mondo One ",/* window title */
                1L,                 /* used for window number */
                0, 0, 0, 0,         /* position when zoomed */
                &W1Ctable,          /* color table */
                0, 0,               /* content vert/horz origin */
                180, 640,           /* height/width of document */
                180, 640,           /* height/width for grow window */
                4, 16,              /* vert/horz pixels for scroll */
                40, 160,            /* vert/horz pixels for page */
                NULL, 0,            /* no info bar */
                NULL,               /* window definition */
                NULL,               /* draw info bar */
                WContent,           /* draw interior */
                40, 100, 159, 540,  /* position / size */
                -1L,                /* plane */
                NULL                /* window record address */
};

ParamList   WindRec2 = {
                sizeof(WindRec2),   /* size of parameter list */
                0xdfa0,             /* frame type */
                "\p Mr. Mondo Two ",/* window title */
                2L,                 /* used for window number */
                0, 0, 0, 0,         /* position when zoomed */
                NULL,               /* color table */
                0, 0,               /* content vert/horz origin */
                180, 640,           /* height/width of document */
                180, 640,           /* height/width for grow window */
                4, 16,              /* vert/horz pixels for scroll */
                40, 160,            /* vert/horz pixels for page */
                NULL, 0,            /* no info bar */
                NULL,               /* window definition */
                NULL,               /* draw info bar */
                NULL,               /* draw interior */
                50, 120, 169, 560,  /* position / size */
                -1L,                /* plane */
                NULL                /* window record address */
};

Boolean hidebit = FALSE;

/* ErrChk()         -- insert from MODEL.C */
/* GetDP()          -- insert from MODEL.C */
/* StartUpTools()   -- insert from MODEL.C */
```

```
/*-----------------------------*
 *  Prepare Desktop and Menus   *
 *-----------------------------*/
PrepDeskTop()
{
    static char *AppleMenu[] = {
        ">>@\\XN1",
        "--About This Program...\\N256",
        "---\\D",
        ">"
    };

    static char *WindowMenu[] = {
        ">>  Window  \\N2",
        "--Open Window1\\N257",
        "--Hide Window1\\DN258",
        "--Close Window1\\DVN259",
        "--Open Window2\\N260",
        "--Close Window2\\DN261",
        ">"
    };

    static char *QuitMenu[] = {
        ">>  Quit  \\N3",
        "--Quit\\N262*Qq",
        ">"
    };

    RefreshDesktop(nil);                       /* Display Desktop */
    InitCursor();                              /* Show mouse cursor */

    InsertMenu(NewMenu(QuitMenu[0]), 0);       /* Install menus */
    InsertMenu(NewMenu(WindowMenu[0]), 0);
    InsertMenu(NewMenu(AppleMenu[0]), 0);

    FixAppleMenu(1);                           /* Display menu bar */
    FixMenuBar();
    DrawMenuBar();
}

/*-----------------------------*
 *  Apple Menu: About          *
 *-----------------------------*/

About()
{
        /* Does nothing (for now) */
}

/*-----------------------------*
 *   Window Content Procedure   *
 *-----------------------------*/

WContent()
{
    MoveTo(0x28, 0x20);
    DrawCString("This is a string inside the window.");
}
```

```
/*------------------------------*
 *  Window Menu: Open1          *
 *------------------------------*/

Open1()
{
    WindPtr1 = NewWindow(&WindRec1);    /* open first window */
    DisableMItem(mOpen1);               /* disable open1 item */
    EnableMItem(mHide1);                /* enable these two... */
    EnableMItem(mClose1);
}

/*------------------------------*
 *  Window Menu: Hide1          *
 *------------------------------*/

Hide1()
{
    if (hidebit) {
        ShowWindow(WindPtr1):
        SetMItemName("\pHide Window1", mHide1);
        hidebit = FALSE;
    } else {
        HideWindow(WindPtr1);
        SetMItemName("\pShow Window1", mHide1);
        hidebit = TRUE;
    }
}

/*------------------------------*
 *  Window Menu: Open2          *
 *------------------------------*/

Open2()
{
    WindPtr2 = NewWindow(&WindRec2);    /* open second window */
    DisableMItem(mOpen2);               /* disable open2 item */
    EnableMItem(mClose2);               /* enable close2 item */
}

/*------------------------------*
 *  Window Menu: Close1         *
 *------------------------------*/

Close1()
{
    CloseWindow(WindPtr1);              /* close this window */
    EnableMItem(mOpen1);                /* enable open1 item */
    DisableMItem(mHide1);               /* disable these two... */
    DisableMItem(mClose1);
}

/*------------------------------*
 *  Window Menu: Close2         *
 *------------------------------*/

Close2()
{
    CloseWindow(WindPtr2);              /* close window 2 */
    EnableMItem(mOpen2);                /* enable open2 item */
    DisableMItem(mClose2);                  /* disable close2 item */
}

CloseW()                        /* close whichever window was clicked */
{
    if (GetWRefCon(EventRec.wmTaskData) == 1)
        Close1();
    else
        Close2();
}

/*------------------------------*
 *      Do Menu Selection       *
 *------------------------------*/

DoMenu()
{
    switch (EventRec.wmTaskData) {
    case mAbout:     About();         break;
    case mOpen1:     Open1();         break;
    case mHide1:     Hide1();         break;
    case mClose1:    Close1();        break;
    case mOpen2:     Open2();         break;
    case mClose2:    Close2();        break;
    case mQuit:      QFlag = TRUE;    break;
    }

    HiliteMenu(FALSE, EventRec.wmTaskData>>16);
}

/* ShutDownTools()  -- insert from MODEL.C */

/*------------------------------*
 *          Main                *
 *------------------------------*/

main()
{
    StartUpTools();              /* Start toolsets */
    PrepDeskTop();               /* Prepare desktop and menus */

    QFlag = FALSE;
    EventRec.wmTaskMask = 0x00001fff;

    while (!QFlag) {
        Event = TaskMaster(0xffff, &EventRec);
        switch (Event) {
        case wInMenuBar:    DoMenu();   break;
        case wInGoAway:     CloseW();   break;
        }
    }

    ShutDownTools();             /* Shutdown all tools started */
    exit(0);
}
```

Program 9-3. Pascal Source for MONDO.PAS

```
( *-----------------------------------------------*
  *                  MONDO.PAS                    *
  *   Desktop Application in TML Pascal (v1.01)   *
  *-----------------------------------------------* )

( **NOTE**  This is not a complete program.  Merge sections
            from the MODEL.PAS program in Chapter Six
            where indicated. )

PROGRAM MondoP;

USES    QDIntF,
        GSIntF,
        MiscTools;

CONST   mAbout   = 256;          ( Menu item IDs )
        mOpen1   = 257;
        mHide1   = 258;
        mClose1  = 259;
        mOpen2   = 260;
        mClose2  = 261;
        mQuit    = 262;

( *------------------------------*
  *      Global Variables        *
  *------------------------------* )

VAR EventRec:   EventRecord;    ( Taskmaster Structure )
    Event:      Integer;        ( Event code )
    UserID:     Integer;        ( Our User ID )
    MemID:      Integer;        ( Memory allocation ID )
    DPBase:     Integer;        ( Direct Page base pointer )
    QFlag:      Boolean;        ( Boolean: Quit flag )

    AppleMenu:  String;         ( Pull down menu strings )
    WindowMenu: String;
    QuitMenu:   String;

    WindPtr1:   WindowPtr;      ( Window port pointers )
    WindPtr2:   WindowPtr;
    W1Title:    String;
    W2Title:    String;

    W1Ctable:   WindowColorTbl;

    WindRec1:   NewWindowParamBlk;
    WindRec2:   NewWindowParamBlk;

    hidebit:    Boolean;

 ( PROCEDURE ErrChk -- insert from MODEL.PAS )
 ( FUNCTION GetD -- insert from MODEL.PAS )
 ( PROCEDURE StartUpTools -- insert from MODEL.PAS )

 ( *------------------------------*
   *  Prepare Desktop and Menus   *
   *------------------------------* )
```

```
PROCEDURE PrepDeskTop;
VAR
    Height:     Integer;        { Menu bar heigth (unused) }

BEGIN
    AppleMenu := CONCAT('>>@\XN1\0',
                        '--About This Program...\N256\0',
                        '---\D\0',
                        '>');

    WindowMenu := CONCAT('>>  Window  \N2\0',
                         '--Open Window1\N257\0',
                         '--Hide Window1\DN258\0',
                         '--Close Window1\DVN259\0',
                         '--Open Window2\N260\0',
                         '--Close Window2\DN261\0',
                         '>');

    QuitMenu := CONCAT('>>  Quit  \N3\0',
                       '--Quit\N262*Qq\0',
                       '>');

    Refresh(Nil);                           { Display Desktop }
    InitCursor;                             { Show mouse cursor }

    InsertMenu(NewMenu(@QuitMenu[1]), 0);   { Install menus }
    InsertMenu(NewMenu(@WindowMenu[1]), 0);
    InsertMenu(NewMenu(@AppleMenu[1]), 0);

    FixAppleMenu(1);                        { Display menu bar }
    Height := FixMenuBar;
    DrawMenuBar;
END;

{ *------------------------------*
  *  Apple Menu: About           *
  *------------------------------* }

PROCEDURE About;
BEGIN
        { Does nothing (for now) }
END;

{ *------------------------------*
  *   Window Content Procedure   *
  *------------------------------* }


PROCEDURE WContent:
BEGIN
    MoveTo($28, $20);
    DrawString('This is a string inside the window.');
END:

{ *------------------------------*
  *  Window Menu: Open1          *
  *------------------------------* }

PROCEDURE Open1;
BEGIN
    W1Title := ' Mr. Mondo One ';
```

```
WITH W1Ctable DO BEGIN
    FrameColor  := $0000;
    TitleColor  := $0f00;
    TBarColor   := $020f;
    GrowColor   := $00f0;
    InfoColor   := $00f0;
END;

WITH WindRec1 DO BEGIN
    param_length    := SIZEOF(NewWindowParamBlk);
    wFrame          := $dfa0;                { frame type }
    wTitle          := @W1Title;             { window title }
    wRefCon         := 1;                    { window number }
    SetRect         (wZoom, 0, 0, 0, 0);     { position when zoomed }
    wColor          := @W1Ctable;            { color table }
    wYOrigin        := 0;                    { content vert origin }
    wXOrigin        := 0;                    { content horz origin }
    wDataH          := 180;                  { height of document }
    wDataW          := 640;                  { width of document }
    wMaxH           := 180;                  { height of grow window }
    wMaxW           := 640;                  { width of grow window }
    wScrollVer      := 4;                    { vert pixels: scroll }
    wScrollHor      := 16;                   { horz pixels: scroll }
    wPageVer        := 40;                   { vert pixels for page }
    wPageHor        := 160;                  { horz pixels for page }
    wInfoRefCon     := LongInt(nil);         { no info bar }
    wInfoHeight     := 0;                    { no info bar }
    wFrameDefProc   := nil;                  { window definition }
    wInfoDefProc    := nil;                  { draw info bar }
    wContDefProc    := @WContent;            { draw interior }
    SetRect         (wPosition, 100, 40, 540, 159);
    wPlane          := -1;                   { plane }
    wStorage        := nil;                  { window record address }
END;

WindPtr1 := NewWindow(WindRec1);     { open first window }
DisableMItem(mOpen1);                { disable open1 item }
    EnableMItem(mHide1);                 { enable these two... }
    EnableMItem(mClose1);
END;

{ *------------------------------*
  *  Window Menu: Hide1          *
  *------------------------------* }

PROCEDURE Hide1;
BEGIN
    IF hidebit THEN BEGIN
        ShowWindow(WindPtr1);
        SetMItemName('Hide Window1', mHide1);
        hidebit := FALSE;
    END
    ELSE BEGIN
        HideWindow(WindPtr1);
        SetMItemName('Show Window1', mHide1);
        hidebit := TRUE;
    END;
END;
```

```
{ *-----------------------------*
  *  Window Menu: Open2         *
  *-----------------------------* }

PROCEDURE Open2;
BEGIN
    W2Title := ' Mr. Mondo Two ';

    WITH WindRec2 DO BEGIN
        param_length    := SIZEOF(NewWindowParamBlk);
        wFrame          := $dfa0;                   { frame type }
        wTitle          := @W2Title;                { window title }
        wRefCon         := 2;                       { window number }
        SetRect         (wZoom, 0, 0, 0, 0);        { position when zoomed }
        wColor          := nil;                     { color table }
        wYOrigin        := 0;                       { content vert origin }
        wXOrigin        := 0;                       { content horz origin }
        wDataH          := 180;                     { height of document }
        wDataW          := 640;                     { width of document }
        wMaxH           := 180;                     { height of grow window }
        wMaxW           := 640;                     { width of grow window }
        wScrollVer      := 4;                       { vert pixels: scroll }
        wScrollHor      := 16;                      { horz pixels: scroll }
        wPageVer        := 40;                      { vert pixels for page }
        wPageHor        := 160;                     { horz pixels for page }
        wInfoRefCon     := LongInt(nil);            { no info bar }
        wInfoHeight     := 0;                       { no info bar }
        wFrameDefProc   := nil;                     { window definition }
        wInfoDefProc    := nil;                     { draw info bar }
        wContDefProc    := nil;                     { draw interior }
        SetRect         (wPosition, 120, 50, 560, 169);
        wPlane          := -1;                      { plane }
        wStorage        := nil;                     { window record address }
    END;

    WindPtr2 := NewWindow(WindRec2);        { open second window }
    DisableMItem(mOpen2);                   { disable open2 item }
    EnableMItem(mClose2);                   { enable close2 item }
END;

{ *-----------------------------*
  *  Window Menu: Close1        *
  *-----------------------------* }

PROCEDURE Close1;
BEGIN
    CloseWindow(WindPtr1);                  { close this window }
    EnableMItem(mOpen1);                    { enable open1 item }
    DisableMItem(mHide1);                   { disable these two... }
    DisableMItem(mClose1);
END;

{ *-----------------------------*
  *  Window Menu: Close2        *
  *-----------------------------* }

PROCEDURE Close2;
BEGIN
    CloseWindow(WindPtr2);                  { close window 2 }
    EnableMItem(mOpen2);                    { enable open2 item }
    DisableMItem(mClose2);                  { disable close2 item }
END;
```

```
PROCEDURE CloseW;              { close whichever window was clicked }
BEGIN
    IF GetWRefCon(WindowPtr(EventRec.TaskData)) = 1 THEN
        Close1
    ELSE
        Close2:
END;

{ *-------------------------------*
  *       Do Menu Selection       *
  *-------------------------------* }

PROCEDURE DoMenu;
BEGIN
    CASE LoWord(EventRec.TaskData) OF
        mAbout:     About;
        mOpen1:     Open1;
        mHide1:     Hide1;
        mClose1:    Close1;
        mOpen2:     Open2;
        mClose2:    Close2;
        mQuit:      QFlag := TRUE;
    END;

    HiliteMenu(FALSE, HiWord(EventRec.TaskData));
END;

{ PROCEDURE ShutDownTools -- insert from MODEL.PAS }

{ *-------------------------------*
  *            Main               *
  *-------------------------------* }

BEGIN
    StartUpTools;              { Start toolsets }
    PrepDeskTop;               { Prepare desktop and menus }

    QFlag := FALSE;
    hidebit := FALSE;
    EventRec.TaskMask := $00001fff;

    REPEAT
        Event := TaskMaster(-1, EventRec);
        CASE Event OF
            wInMenuBar: DoMenu;
            wInGoAway:  CloseW;
        END;
    UNTIL QFlag;

    ShutDownTools              { Shutdown all tools started }
END.
```

Chapter Summary

The following tool set functions were referenced in this chapter.

Function: $020E
Name: WindStartUp
Starts the Window Manager
Push: UserID (W)
Pull: Nothing
Errors: None

Function: $030E
Name: WindShutDown
Shuts down the Window Manager
Push: Nothing
Pull: Nothing
Errors: None

Function: $090E
Name: NewWindow
Creates a window on the DeskTop
Push: Result Space (L); Window Record (L)
Pull: Window Pointer (L)
Errors: $0E01, $0E02

Function: $0B0E
Name: CloseWindow
Closes a window, removing it from the DeskTop
Push: Window Pointer (L)
Pull: Nothing
Errors: None

Function: $120E
Name: HideWindow Hides a window, making it invisible
Push: Window Pointer (L)
Pull: Nothing
Errors: None

Function: $130E
Name: ShowWindow
Displays a previously hidden window
Push: Window Pointer (L)
Pull: Nothing
Errors: None

Function: $1D0E
Name: TaskMaster
Tracks mouse, menu, and window events
Push: Result Space (W); Event Mask (W); Event Record (L)
Pull: TaskCode (W)
Errors: $0E03

Function: $29OE
Name: GetWRefCon
Returns the value of a window wRefCon parameter
Push: Result Space (L); Window Pointer (L)
Pull: Window's wRefCon (L)
Errors: None

Menu Item Calls

Function: $300F
Name: EnableMItem
Enables a dimmed menu item
Push: Menu Item's ItemNum (W)
Pull: Nothing
Errors: None

Function: $310F
Name: DisableMItem
Dims, or disables, a menu item
Push: Menu Item's ItemNum (W)
Pull: Nothing
Errors: None

Function: $3A0F
Name: SetMItemName
Changes the name of a menu item
Push: Pascal String (L); Menu Item's ItemNum (W)
Pull: Nothing
Errors: None

QuickDraw II Calls

Function: $3A04
Name: MoveTo
Moves the graphics pen to a specific coordinate
Push: Horz Position (W); Vert Position (W)
Pull: Nothing
Errors: None

Function: $A604
Name: DrawCString
Displays a C string in graphics mode
Push: C String (L)
Pull: Nothing
Errors: None

Chapter 10

Dialog Boxes

Dialog boxes offer you a chance to communicate with the person using your program. Like the buttons and viewing window on the front of an automatic teller machine, dialog boxes are the most easily understood ways for a computer to display information and obtain input, particularly when

compared to the old-fashioned Yes/No prompts and dreary command line options.

This chapter covers the Dialog Manager and the creation of dialog boxes. Background information is provided initially, with descriptions of the different types of dialog boxes:

- Modal dialog boxes
- Modeless dialog boxes
- Alerts

This chapter also covers the items associated with dialog boxes and all their structures, options, and settings. This is followed by numerous programming examples and explanations. Unlike previous chapters, this chapter does not contain a complete programming example, though you can merge the About. . . dialog box example at the end of this chapter with the MODEL program introduced in Chapter 6.

> Chapter 11, which is about controls, adds a little more information to what's offered here. If you're interested in creating custom dialog boxes with your own controls, it's recommended that you read Chapter 10 first, then Chapter 11.

Background Information

Dialog boxes are controlled by the Dialog Manager. But actually, more than any other tool set, the Dialog Manager relies on a number of other tool sets to help get the job done. For example, from the previous chapter, you might have read that the Window Manager contributes to the Dialog Manager by drawing the actual dialog box. Also, the Control Manager (covered in the next chapter) helps out by drawing, manipulating, and regulating the controls in a dialog box.

To use dialog boxes in your programs, you'll need to have started the following tool sets:

- Tool Locator
- Memory Manager
- Miscellaneous tool set
- QuickDraw II
- Event Manager

- Window Manager
- Control Manager
- LineEdit tool set

(Also refer to the table of tool set dependencies in Chapter 4.)

It may seem rather strange that the LineEdit tool set is required to use a dialog box. In fact, you cannot display any text in a dialog box unless you've started the LineEdit tool set. The main reason LineEdit is needed is to manipulate text in a text input box (EditLine item).

The text input box, as well as numerous other goodies you can put into a dialog box, are covered in Chapter 11, which deals with controls.

The Dialog Manager is started by a call to the DialogStartUp function and shut down by a call to the DialogShutDown function. The Dialog Manager shares direct page space with the Control Manager, so there's no need to specify direct page space when starting this tool set.

In machine language, the following code can be used to start the Dialog Manager (remember that the above-mentioned tool sets should also have been started):

```
pushword        UserID    ;push the program's User ID
_DialogStartUp            ;No errors possible
```

In C and Pascal:

```
DialogStartUp(UserID);
```

To avoid compile-time errors, C programmers should note that the <dialog.h> header file should be included at the top of your program along with the header files for all the other tool sets that are started up.

To shut down the Dialog Manager, the following routines can be used.

In machine language:

```
_DialogShutDown
```

In C:

```
DialogShutDown( );
```

And in Pascal:

```
DialogShutDown;
```

Once the Dialog Manager is started, your program can display dialog boxes. The dialog boxes can be defined in three ways, using three separate, yet similar, Toolbox calls. Once the dialog box is activated, there are special Dialog Manager calls that monitor the events in the dialog box. All these techniques, including examples of several dialog boxes, are described below.

Types of Dialog Boxes

As was mentioned earlier in this chapter, there are three types of dialog boxes:

- Modal
- Modeless
- Alert

Modal. A modal dialog box is the most common traditional type of dialog box. It's typically a rectangle filled with controls or a message. The dialog box is where a dialogue can take place between the user and the program. A model dialog box allows the user to set or change an option or it can simply display information as in an About. . . or a Help dialog box.

Modeless. The modeless dialog box is the least understood of the three. It's basically a window with dialog controls in it. Unlike the modal dialog box, which is always the foremost window, a modeless dialog box can be placed behind other windows, moved, zoomed, or manipulated like a regular window. Because of this extra activity, the modeless dialog boxes are a little harder to program. Also, their use is vaguely defined, so you won't see them very often.

Modal? Modeless? How can you remember which one does what?

A good question. Think of a modal dialog as one that puts you in a *mode* where you're essentially forced to interact only with that dialog. A modeless dialog box is one without such restrictions: It's present on the DeskTop, but doesn't force you to interact with it.

Alert. The third type of dialog box, the alert, displays a warning and, to varying degrees, a message. Alerts can have OK/Continue or Cancel/Stop buttons in them. The alert dialog boxes are actually specialized forms of modal dialog boxes.

Refer to the Human Interface Guidelines Appendix for more information on the use of the dialog box as well as for design guidelines.

Creative Overview

Dialog boxes are easy to use. About the hardest thing they require is that you organize your thoughts about what to put into them. Utilizing a combination of tool set functions, the Dialog Manager simplifies the monitoring of dialog box events. Your program acts upon those events and performs whatever actions are necessary.

Dialog boxes, like windows, require tables, locations, pointers, strings—a lot of information. In fact, positioning the controls is the only difficult thing about doing one. You'll spend more time making minor adjustments in the way things are displayed than you will placing them into the dialog box, or debugging logic.

The steps to building a standard, modal dialog box are as follows:

1. Define the dialog box.
2. Place items into the dialog box.
3. Wait for a dialog event.
4. Act on the event (repeat steps 3 and 4 as needed).
5. Close the dialog box when you've finished.

Steps 3 and 4 are repeated as various options in the dialog box are set. According to the Human Interface Guidelines, at least one button in the dialog should be responsible for closing the dialog box and making it go away. Typically, two buttons, OK and Cancel, are used for this purpose.

Actually, a dialog box could contain only a text message such as the famous saying, *Please wait while I initialize.* As soon as the program was ready, it could remove the dialog box and then proceed.

In step 1, the dialog box is defined. It is placed on the screen as a special type of window, in front of all other windows on the screen. There are a number of calls to create the different types of dialog boxes. In fact, there are three separate Toolbox calls used to create a standard modal dialog box (each is covered later in this chapter).

After the dialog box is created (by whichever method), the Dialog Manager returns a pointer used to further reference the dialog box, just as the Window Manager returns a pointer to a window. The pointer returned by the Dialog Manager is used to place items into that particular dialog box, as well as to remove the dialog box once you've finished with it.

Step 2 is where items are placed into the dialog box. Each item has a position relative to the top left corner of the dialog box (local coordinates), an item description, and a type. The individual characteristics of the items, or controls, placed into a dialog box are covered in the next section.

Steps 3 and 4 are where all the activity takes place. The Dialog Manager has special functions that monitor dialog box activity. These functions take advantage of the TaskMaster and Event Manager to make tracking the events in a dialog box quite simple. When a user selects a particular control, your program can determine which control was selected and take appropriate action.

Once the user has finished with the dialog box (OK or Cancel has been clicked), the dialog box is closed, just like a window. The dialog box can be called up again a number of times by simply repeating these steps. See below for individual examples of how these steps are implemented.

Modal dialog boxes will, without exception, follow the above five steps. Alert boxes are special exceptions. With alerts, the first three steps are combined (most of the work is done internally, by the Toolbox). Alerts are used only to get an immediate yes/no response from a user. Therefore no additional action is taken upon them. They are first displayed; then they get the input and are finally removed so that your program can continue with the action or stop what it's doing. (See the section on alerts below for more information.)

Modeless dialog boxes are handled in a completely different manner. A modeless dialog box is displayed; however, unlike modal dialog boxes and alerts, it need not be acted upon right away. The user can move it behind other windows on the DeskTop, or ignore it completely and go off to do something else. Because of this, modeless dialog boxes have a special way of handling their events. (Refer to the section on modeless dialog boxes below for additional information.)

Dialog Box Controls

The things placed into a dialog box are called controls. Buttons are a common type of control, as are radio buttons, check boxes, text input boxes (EditLines), pictures, icons, and even blocks of text.

Every control placed into a dialog box has a special ID number associated with it. It's this value that is monitored by the Dialog Manager's special event-handling routines (step 3 from the previous section). When the user clicks on that control, the ID number is returned for your program to examine. Simple.

Besides assigning an ID number, you also need to define what type of item is placed into your dialog box, where it is placed, whether it's visible, invisible, disabled, and so forth. In all, you need to tell the Dialog Manager seven things in order to place a control into a dialog box (see Table 10-1).

Table 10-1. Seven Parameters for Placing Controls in Dialog Boxes

Name	Value	Meaning
ItemID	Word	The control's special ID number
ItemRect	Rectangle	The control's position inside the dialog
ItemType	Word	The type of control: button, text, icon, and so on
ItemDescr	Long	A pointer to special information about the control
ItemValue	Word	The initial value of a control
ItemFlag	Word	Visible/invisible flag, as well as other information
ItemColor	Pointer	A table defining the dialog's color

These items are placed into the dialog box either individually—by using the NewDItem Toolbox call—or all at once—by using a template of information, or record, and using the GetNewDItem or GetNewModalDialog calls.

Individually, each item is described as follows.

ItemID. The ItemID is a value assigned to each control in your dialog box. It can be any value in the range $0001–$FFFF. (An Item ID of 0 is possible, but not recommended, because of potential conflicts with certain Toolbox calls.)

An ItemID of 1 is reserved for use by the dialog's default button. Pressing the Return key is considered the same as clicking on the item with an ItemID of 1. Typically, the OK button is given an

ID of 1. Also, if a button has an ID of 1, that button has a double outline.

An ItemID of 2 is reserved for the dialog's Cancel button. Pressing the Escape key is the same as selecting the item in a dialog box with an ID of 2.

Feel free to give the items in your dialog box any number other than 1 and 2 (and 0). A good technique is to give each dialog box an ID in the MSB of the ItemID, then number the controls sequentially starting with 0.

For example, assume your dialog box is given the arbitrary value $0055. Then assign each control in the dialog box (except the OK and Cancel buttons) with IDs of $5500 plus the sequential value of the specific button. Refer to the programming samples below for examples.

> The default button, ItemID $0001, is a good thing to have in any dialog box, especially when you're first writing routines and experimenting. Because pressing the Return key is the same as clicking the default button, if you ever make a terrible formatting mistake (like creating a tall, skinny dialog box with no visible text or controls), you can still press Return to avoid having to reset your computer to start over. This might not exactly be the intent of the default button, but by trial and error, most programmers discover this technique. The authors have become very adept at this.

ItemRect. The ItemRect defines the control's position relative to the upper left corner of the dialog box (which is local coordinate 0,0). The ItemRect is defined as four words setting the upper left corner and lower right corner of the control's location as follows:

- Upper Left Y value (MinY)
- Upper Left X value (MinX)
- Lower Right Y value (MaxY)
- Lower Right X value (MaxX)

Any text in your dialog box must fit inside the given rectangle. If you make that rectangle too small, not all the text will be visible. And if you make the rectangle too large, the text might overlay other controls in the dialog box.

With some controls, such as buttons, you need only define the upper left coordinate, using a value of 0 for the lower right coordinate. The lower right values are calculated based on the size of the text inside the control. (This calculation is performed automatically by the Control Manager.) For example,

```
dc  i2'70,130,0,0'
```

is all right to define the location of a button. The MaxY and MaxX values are set according to the text in the button.

> In machine language and C, the values of a rectangle are given in MinY, MinX, MaxY, MaxX order. But in *TML Pascal*, coordinates use the MinX, MinY, MaxX, MaxY order. Keep this in mind when converting programs between these languages.

ItemType. The ItemType parameter describes the type of control. ItemTypes in the following table are listed next to the items they define.

Table 10-2. ItemTypes and the Control They Describe

ItemType	Description	Name	Definition
$000A	Button	ButtonItem	Activator
$000B	Check box	CheckItem	Switch
$000C	Radio button	RadioItem	Switch
$000D	Scroll bar	ScrollBarItem	Special dialog control
$000E	User control	UserCtlItem	User-defined
$000F	Text	StatText	Characters (up to 255)
$0010	Text (longstat)	LongStatText	Characters (up to 32,767)
$0011	EditLine	EditLine	Input box
$0012	Icon	IconItem	Graphic image
$0013	Picture	PicItem	Graphic image
$0014	User item	UserItem	User-defined
$0015	User control 2	UserCtlItem2	User-defined

Currently, only the above ItemTypes are defined. So, for example, to define a check box in your dialog, you'd specify an item type of $000B (as well as providing the other information indicated in this section).

To disable any item in the dialog box (so that clicking the mouse on that item will not generate a dialog event), logically OR the ItemType with $8000 (which is the same as adding $8000 to the item value). For example, most text items in a dialog box are disabled, meaning that clicking on them doesn't do anything. To define a disabled text item, the following ItemType can be used:

```
dc  12'$800F'
```

This might also be expressed using equates in machine language (see the examples below), as in

```
dc  12'ItemDisable+StatText'
```

where ItemDisable equals $8000 and StatText equals $000F.

In C, the expression

```
(itemDisable | statText)
```

is equivalent to adding these two items, though more logical.

ItemDescr. The ItemDescr is a long word, either a pointer or a handle, depending on the ItemType (see Table 10-3).

Table 10-3. ItemType Determines What Is Pointed to by ItemDescr

ItemType	ItemDescr
Picture	Picture's handle
Button	Pointer to a string to be placed inside the button
Check box	Pointer to the check box's title string
Radio button	Pointer to the radio button's title string
Scroll bar	Pointer to an action procedure controlling a scroll bar
User control	Pointer to the control's action procedure
Text	Pointer to the text string
Text (longstat)	Pointer to the beginning of the block of text
EditLine	Pointer to a text string or buffer
Icon	Icon's handle
User item	Pointer to a definition procedure
User control 2	Pointer to a parameter block

All string pointers above indicate the memory location of a Pascal-type string.

ItemValue. The ItemValue of a control contains the control's initial value, or 0 in most cases (see Table 10-4).

Table 10-4. Values Contained in ItemValue

ItemType	ItemValue
Picture	Pointer to the picture's image
Button	Initial value of the control
Check box	$0001 to check the box, $0000 for unchecked
Radio button	$0001 to fill the button, $0000 to leave it empty
Scroll bar	Value passed to the scroll bar's definition procedure
Text	Not important
Text (longstat)	Number of characters in the text block (up to 32,767)
EditLine	Maximum number of characters to be entered (up to 255)
Icon	Not important
User item	Initial value of the control
User control 2	Initial value of the control

The value can be examined or changed using the Dialog Manager Toolbox calls GetDItemValue and SetDItemValue. For example, suppose a radio button is to be activated based upon some change in the program. The following routines will change the ItemValue of the radio button.

In machine language:

```
ResetRB     anop
            pha                                ;push one word result space
            pushlong        DialogPtr          ;the pointer to the dialog box
            pushword        #RButton1          ;the ItemID of the radio button
            _GetDItemValue                     ;return its value

            pla                                ;test the item's current value
            bne             Go_On              ;if it's already 1, don't change it

            pushword        $0001              ;the new value for the item (1=on)
            pushlong        DialogPtr
            pushword        #RButton1
            _SetDItemValue                     ;set the new value

Go_On       anop
```

Note: GetDItemValue and SetDItemValue return an error ($150C) if the ItemID specified does not exist or does not belong to the specified dialog box.

In C:

```
if (!GetDItemValue(DialogPtr, RButton1))
  SetDItemValue(1, DialogPtr, RButton1);
```

In Pascal:

```
IF GetDItemValue(DialogPtr, RButton1) = 0 THEN
   SetDItemValue(1, DialogPtr, RButton1);
```

Note: Clicking on a radio button or check box does not automatically activate it. Your program must do that.

The value of the radio button can also be set when the dialog box is initially created. However, the above routines are preferred if the state of the radio button changes. See the COLOR example below. Also, be careful not to confuse changing the ItemValue with making it invisible or disabling it.

ItemFlag. The ItemFlag is used mainly by the Control Manager to control certain aspects of some controls—for example, the outline of a button or the orientation of a scroll bar. Refer to Chapter 11 for information on the ItemFlag. For now, setting ItemFlag to 0 in your routines is acceptable.

ItemColor. ItemColor is a long word that points to a color table. The color table is used by the Control Manager to change the colors of the item. Normally, this item is set to a long word of 0 and the standard colors are used. Refer to Chapter 11, which deals with controls, for a description of the color table and an example of changing an item's color.

A Dialog Box

There are three "official" methods for placing a modal dialog box on the screen. The first one is the most complex. It pushes all information about the dialog box on the stack, then calls the Dialog Manager a number of times (once to create the dialog, then one time for each item in the dialog) until everything's finished.

The other two methods use templates of information. These templates merely contain all the data that is pushed to the stack in the first method. However, with templates, only a pointer to a template, or simply to one master template, is pushed to the stack. The Dialog Manager does the rest.

The complex method of creating a dialog is covered in this section, along with important background information. The methods using the templates appear in the following two sections.

To create a dialog box, you must tell the Dialog Manager the following three things:

- The location and size of the dialog box
- Whether the dialog box is visible or not
- A long word value, DRefCon

The DRefCon is a value your program can define for its own use. As with the wRefCon value used by the Window Manager to define a window (see Chapter 9), this value is typically set to 0, but it can be set to any value you'd like.

From the Dialog Manager your program will receive a long-word pointer to the dialog's port, or a long word of 0 if there was an error. This value should be saved for all further references to your dialog box.

Once the dialog is established, you can start placing controls into it. As with creating a dialog box, the controls can be created by pushing their values on the stack and calling a Dialog Manager routine to install them one at a time, or you can use templates to install them all at once.

Simply creating and placing the dialog items does not make them appear in the dialog box. They all suddenly appear the first time you make a call to the ModalDialog function—which is a good thing, because that's what your program will use to handle dialog events.

When the desired controls have been placed into the dialog box, the ModalDialog function handles dialog events, just as the Event Manager or TaskMaster handles desktop events. ModalDialog also initially places all the items into the dialog box. (The items are not visible until ModalDialog is called.)

The ModalDialog function is used only for modal dialog boxes. Modeless dialog boxes and alerts use their own methods for trapping dialog box events. These techniques are discussed in a later section.

ModalDialog waits for the user to click the mouse on a control. When this happens, the ItemID of the control is returned by the ModalDialog function, even for EditLine items. Your program can then take whatever action is necessary.

Once the function of the dialog box is served, close it, removing it from the screen, with the CloseDialog function.

It's important to include some way to close a dialog box. In other words, build in an option for the user to tell the dialog box to go away. It's embarrassing when professional programmers and gurus create magnificent dialog boxes and then realize they have no way of escaping from them.

Important Pascal Note

At the time of this writing some important *TML Pascal* data types for the Dialog Manager had not been finalized. So, for your programming pleasure, a set of records and data types are listed next. These are all related to working with dialog boxes and alert boxes in Pascal, and they are used throughout the examples in the rest of this book. You can incorporate this information into your programs as needed.

Note that the remainder of this book refers to these types as if they were automatically built into a *TML Pascal* unit symbol file. The definitions of these types won't be shown again.

```
{ TML Pascal Dialog and Alert Type Definitions }
CONST   atItemListLength = 4;
        dtItemListLength = 8;
TYPE    ItemTempPtr = ^ItemTemplate;
        ItemTemplate = PACKED RECORD
                                              ItemID:        Integer;
                                              ItemRect:      Rect;
                                              ItemType:      Integer;
                                              ItemDescr:     Ptr;
                                              ItemValue:     Integer;
                                              ItemFlag:      Integer;
                                              ItemColor:     Ptr;
        END;
DialogTemplate = RECORD
        dtBoundsRect: Rect;
        dtVisible: Boolean;
        dtRefCon: Longint;
        dtItemList: ARRAY [0..dtItemListLength] OF ItemTempPtr;
END;
AlertTempPtr = ^AlertTemplate;
AlertTemplate = RECORD
        atBoundsRect: Rect;
        atAlertID: Integer;
        atStage1: SignedByte;
        atStage2: SignedByte;
        atStage3: SignedByte;
        atStage4: SignedByte;
        atItemList: ARRAY [0..atItemListLength] OF ItemTempPtr;
END;
```

Check your version of *TML Pascal* to see whether these types (or similar types) are defined. If they are, the names might be different. (The authors did their best to choose record and field names that seemed the most logical, but they're not clairvoyant.)

Doing a Dialog, the Long Way

The first Toolbox function used to create a dialog box is NewModalDialog. It receives its information on the stack rather than using a template. The following routines can be used to create a modal dialog box using the NewModalDialog function.

In machine language:

```
LadyDi      pha                                      ;long word result space
            pha
            pushlong          #DialogRect            ;rectangle pointer
            pushword          TRUE                   ;make dialog visible (TRUE =
                                                     ;$8000)
            pea               $0000                  ;DRefCon - any value
            pea               $0000
            _NewModalDialog                          ;make the call
            jsr               ErrChk                 ;check for errors

            pulllong          DialogPtr              ;the dialog pointer
            rts

DialogRect  dc                i2'40,30,100,290'      ;its position and size (320 mode)
```

In C:

```
Rect DialogRect = { 40, 30, 100, 290 };
LadyDi( )
{
   DialogPtr = NewModalDialog(&DialogRect, TRUE, NULL);
}
```

In Pascal:

```
PROCEDURE LadyDi;
VAR DialogRect : Rect;
BEGIN
    SetRect(DialogRect, 30, 40, 290, 100);
    DialogPtr := NewModalDialog(DialogRect, TRUE, LongInt(nil));
END;
```

The size and position of the dialog box are specified by the rectangle passed to the NewModalDialog function. According to the Human Interface Guidelines, dialog boxes should be a little higher than screen center and centered left to right. The following machine language equations can be used to center a dialog box. The DHeight and DWidth parameters represent the dialog box's height (Y pixels) and width (X pixels), respectively.

```
DHeight     equ  ??                          ;Your dialog's height goes here
DWidth      equ  ??                          ;Your dialog's width goes here

DialogRect  dc   i2'(190-DHeight)/2'
            dc   i2'(640-DWidth)/2'
            dc   i2'(190-DHeight)/2+DHeight'
            dc   i2'(640-DWidth)/2+DWidth'
```

For a dialog box in the 320 screen mode, change the number 640 above to 320. The value 190 is used for the maximum number of Y pixels to place the dialog box a little above center screen. (It looks awkward when a value of 200 is used.)

This technique can be used in your programs as needed, either as a pointer or as part of a dialog's template (see below). Remember to replace the DHeight and DWidth values in the template with the equates (or values) representing the size of your particular dialog box.

Items placed inside the dialog box are given in local coordinates relative to the upper left corner of the dialog (position 0,0). This allows you to move or resize the dialog box without affecting the internal location of the items.

Items inside a dialog box are placed there by a call to the Dialog Manager's NewDItem function. NewDItem requires the information listed in Table 10-5 for the item you're placing into the dialog.

Table 10-5. Information Required by NewDItem

Parameter	Size	Description
DialogPtr	Long	A pointer to the dialog box
ItemID	Word	The control's ID number
ItemRect	Long	A pointer to the control's position
ItemType	Word	The type of control
ItemDescr	Long	A pointer to special information about the control
ItemValue	Word	The control's initial value
ItemFlag	Word	Miscellaneous information about the control
ItemColor	Long	A pointer to the control's color table

In the following programming examples, the first control defined is the OK button; the second control, a block of text.

In machine language:

```
ButtonItem    equu         $000A
TextItem      equ          $000F
ItemDisable   equ          $8000
TrickDi       pushlong     DialogPtr               ;dialog in which to place this control
              pushword     $0001                   ;the ItemID, 1 = default button
              pushlong     #ButtonRect             ;rectangle pointer for the button
              pushword     ButtonItem              ;this item is a button, type $000A
              pushlong     #ButtonText             ;text inside button
              pushword     $0                      ;initial value (not important)
              pushword     $0                      ;itemFlag, zero for default
              pushlong     $0                      ;color table, zero for default
              _NewDItem                            ;make the call
              jsr          ErrChk                  ;check for errors

              pushlong     DialogPtr               ;second item: text block
              pushword     $1234                   ;ItemID, can be anything
              pushlong     #TextRect
              pushword     TextItem+ItemDisable
              pushlong     #TextText
              pushword     $0
              pushword     $0
              pushlong     $0
              _NewDItem
              jmp          ErrChk
ButtonRect    dc           12'35,150,0,0'          ;its position in the dialog (relative)
                                                   ;ButtonText
str           'OK'         ;button's text
TextRect      dc           12'10,60,30,240'
TextText      str          'Press the OK button'
```

In C:

```
Rect ButtonRect = { 35, 150, 0, 0 };
Rect TextRect = { 10, 60, 30, 240 };

TrickDi()
{
   NewDItem(DialogPtr, 1, &ButtonRect, buttonItem,
               "\pOK", 0, 0, NULL);
   ErrChk();
   NewDItem(DialogPtr, 0x1234, &TextRect, textItem+itemDisable,
               "\pPress the OK button", 0, 0, NULL);
   ErrChk();
}
```

In Pascal:

```
PROCEDURE TrickDi;
VAR ButtonRect : Rect;
   TextRect : Rect;
   ButtonText : String;
   TextText : String;
BEGIN
   ButtonText := 'OK';
   TextText := 'Press the OK button';
   SetRect(ButtonRect, 150, 35, 0, 0);
   SetRect(TextRect, 60, 10, 240, 30);
   NewDItem(DialogPtr, 1, ButtonRect, ButtonItem,
               @ButtonText, 0, 0, nil);
   ErrChk;
   NewDItem(DialogPtr, $1234, TextRect, StatTextItem+ItemDisable,
               @TextText, 0, 0, nil);
   ErrChk;
END;
```

Once all the controls have been placed in the dialog box, the ModalDialog function is called to monitor dialog events. ModalDialog returns the ItemID of the control selected with the mouse. The following routines incorporate the previous two examples to monitor the pressing of the OK button. When OK is pressed, the dialog box is closed via the CloseDialog function.

In machine language:

```
Wait    pha                             ;one word result space
        pushlong        DialogPtr       ;this dialog
        _ModalDialog                    ;make the call

        pla                             ;get results, the ItemID
        cmp             #$1             ;was it OK?
        bne             Wait            ;keep waiting if not OK

        pushlong        DialogPtr       ;close this dialog
        _CloseDialog                    ;do it
```

In C:

```
while (ModalDialog(DialogPtr) != 1);
CloseDialog(DialogPtr);
```

In Pascal:

```
REPEAT UNTIL (ModalDialog(DialogPtr) = 1);
CloseDialog(DialogPtr);
```

Once the dialog is closed, the Event Manager/TaskMaster continues monitoring your DeskTop events. The dialog can again be opened to obtain input, adjust settings, or communicate a message, simply by repeating the above steps.

Making It Easier

The only thing wrong with the routines in the previous section is that they involve a lot of typing (especially in machine language). When you replace the information pushed to the stack with templates of information, the actual code used to create the dialog box becomes easier to read. Also, updating the dialog box is easier because you're changing data templates rather than changing actual program code.

To add a control to a dialog box using templates, the GetNewDItem function is used. GetNewDItem does the same thing as NewDItem, except the information is in a template, and a long pointer to that template is passed to the Toolbox. Refer to Table 10-6 for details about the structure of the template.

Table 10-6. Structure of GetNewDItem Template

Offset	Size	Parameter
+$00	Word	ItemID
+$02	Four words (rectangle)	ItemRect
+$0A	Word	ItemType
+$0C	Long word (pointer)	ItemDescr
+$10	Word	ItemValue
+$12	Word	ItemFlag
+$14	Long word (pointer)	ItemColor

The following routines are similar to those found in the previous section. They define the same two controls—a button and a block of text—using the GetNewDItem function.

In machine language:

```
ButtonItem    equu       $000A
TextItem      equ        $000F
ItemDisable   equ        $8000

PutItems      anop
              pushlong   DialogPtr          ;dialog in which to
                                            ;place this control
              pushlong   #ButtonRec         ;the button's template
              _GetNewDItem
              jsr        ErrChk             ;test for errors
```

```
            pushlong        DialogPtr               ;dialog in which to
                                                    ;placs this control
            pushlong        #TextRsc                ;the text's template
            _GetNewDItem
            jmp             ErrChk
                                                    ;the button's template
ButtonRec   anop
            dc              i2'1'                   ;ItemID, 1
            dc              i2'35,150,0,0'          ;rsctangls for the button
            dc              i2'ButtonItsm'          ;ItemTyps
            dc              i4'ButtonText'          ;pointer to button's text
            dc              i2'0'                   ;initial value (not

                                                    ;important)
            dc              i2'0'                   ;itemFlag, zero for

                                                    ;default
            dc              i4'0'                   ;color tabls, default
ButtonText  str             'OK'                    ;button's text
                                                    ;ths tsxt's tsmplats
TextRec     anop
            dc              i2'$1234'
            dc              i2'10,60,30,240'
            dc              i2'TextItem+ItemDisable'
            dc              i4'TextText'
            dc              i2'0'
            dc              i2'0'
            dc              i4'0'
TextText    str    'Press the OK button'
```

In C:

```
ItemTemplate    ButtonRec = {
        1,                      /* ItemID = 1 */
        35, 150, 0, 0,          /* rectangle for the button */
        buttonItem,             /* ItemType */
        "\pOK",                 /* button'e text */
        0,                      /* ItemValue */
        0,                      /* ItemFlag */
        NULL                    /* color table, default */
};
ItemTemplate    TextRec = {
        0x1234,
        10, 60, 30, 240,
        textItem + itemDieable,
        "\pPress the OK button",
        0,
        0,
        NULL
};

PutItems( )
{
        GetNewDItem(DialogPtr, &ButtonRec);     ErrChk( );
        GetNewDItem(DialogPtr, &TextRec);       ErrChk( );
}
```

In Pascal:

```
PROCEDURE PutItems;
VAR     ButtonRec :             ItemTemplate;
        TextRec :               ItemTemplate;
        TextText :              String;
        ButtonText :            String;
BEGIN
        TextText                                := 'Press the OK button';
        ButtonText                              := 'OK';

        WITH ButtonRec DO BEGIN
              ItemID                            := 1;
              SetRect (ItemRect, 150, 35, 0, 0);
              ItemType                          := ButtonItem;
              ItemDescr                         := @ButtonText;
              ItemValue                         := 0;
              ItemFlag                          := 0;
              ItemColor                         := nil;
        END;

        WITH TextRec DO BEGIN
              ItemID                            := $1234;
              SetRect(ItemRect, 60, 10, 240, 30);
              ItemType                          := ItemDisable+StatTextItem;
              ItemDescr                         := @TextText;
              ItemValue                         := 0;
              ItemFlag                          := 0;
              ItemColor                         := nil;
        END;

        GetNewDItem(DialogPtr, ButtonRec);    ErrChk;
        GetNewDItem(DialogPtr, TextRec);      ErrChk;
END;
```

The other routines from the previous section, NewModalDialog and ModalDialog (for dialog box event trapping), would still be used as written. The GetNewDItem only aids in the creation of controls.

Do you get the feeling that perhaps you should have started to read this chapter from the end and then worked backwards?

The final step to creating a dialog box easier is just to use one big template for everything—that is, for the dialog box as well as all the controls in the dialog box. This way, creating a dialog box with all its goodies is done with just one Dialog Manager Toolbox call: GetNewModalDialog.

> GetNewModalDialog would seem to be the longest-named Toolbox function. Well, it is. At 17 letters, it ties with FFSoundDoneStatus and TLTextMountVolume. GetNewModalDialog works internally by calling NewModalDialog and then GetNewDItem for each item in the template.

The template used by GetNewModalDialog contains the information listed in Table 10-7. (Note how it also incorporates the templates used by GetNewDItem.)

Table 10-7. Information Required by GetNewModalDialog Template

Offset	Size	Parameter	Description
+$00	Four words (rectangle)	BoundsRect	Size/location of dialog box
+$08	Word	dtVisible	Visible/invisible flag
+$0A	Long word	dtRefCon	Whatever you want
+$0E	Long word (pointer)	ItemPtr	First item's template
+$12	Long word (pointer)	ItemPtr	Second item's template (and so on)
+$??	Long word	Terminator	Zero, end of template

The dialog box's template contains all the information passed to the NewModalDialog function, as well as pointers to the control item's templates used by the GetNewDItem function. The last item in the dialog box's template is a long word of 0 to indicate the end of the template. This way, your dialog box can have a multitude of items (though that's not recommended). See the COLOR example below for a really huge template example.

Incorporating all the information from the previous two sections, the following examples create the dialog box and place all those items into the dialog box. Use the ButtonRec and TextRec data from the examples in the previous section to complete the examples below.

In machine language:

```
PutItems    anop
            pha                                 ;long word result space
            pha
            pushlong           #DialogRec       ;dialog's template
            _GetNewModalDialog                  ;do it all!
            jsr                ErrChk           ;check for errors
            pulllong           DialogPtr        ;the dialog pointer
            rts
DialogRec   anop                                ;dialog's template
            dc                 i2'40,30,100,290' ;dialog's rectangle
            dc                 i2'TRUE'         ;visible flag
            dc                 i4'0'            ;DRefCon - any value
            dc                 i4'ButtonRec'    ;first control's template
            dc                 i4'TextRec'      ;second control's template
            dc                 i4'0'            ;null terminator
```

In C:

```
DialogTemplate DialogRec = {
                          40, 30, 100, 290,  /* dialog's rectangle */
                          TRUE,              /* visible flag */
                          NULL,              /* dtRefCon */
                          &ButtonRec,        /* first control's template */
                          &TextRec,          /* second control's template */
                          NULL               /* null terminator */
};
PutItems()
{
   DialogPtr = GetNewModalDialog(&DialogRec);
}
```

In Pascal:

```
PROCEDURE
PutItems;
VAR DialogRec : DialogTemplate;
BEGIN
            WITH DialogRec DO BEGIN
                        SetRect(dtBoundsRect, 30, 40, 290, 100);
                        dtVisible       := TRUE;
                        dtRefCon        := Longint(nil);
                        dtItemList[0]   := @ButtonRec;
                        dtItemList[1]   := @TextRec;
                        dtItemList[2]      := nil;
            END;
            DialogPtr           := GetNewModalDialog(&DialogRec);
END;
```

Following the above routines, your program should monitor the dialog events with the ModalDialog function and, when finished, close the dialog box with the CloseDialog function. Unfortunately, there are no simple shortcuts for those two calls. (After all, they really are simple themselves.)

The list of ItemTemplate pointers in C is actually an array which has eight elements allocated. If your program has a dialog box that contains more than eight items, you'll have to increase the size of that array to handle more elements. This is done by defining a constant dtItemListLength. It should be placed before the #include <dialog.h> directive at the top of your program—for example:

```
#define dtItemListLength 14    /* define a larger item array */
#include <dialog.h>
```

Pascal programmers need only change the dtItemListLength constant in the CONST section of their programs.

Alert Boxes

An alert box is a special type of dialog box. It's used to display a message and usually offers two buttons:

- One to go on (OK)
- One to stop whatever action is taking place (Cancel)

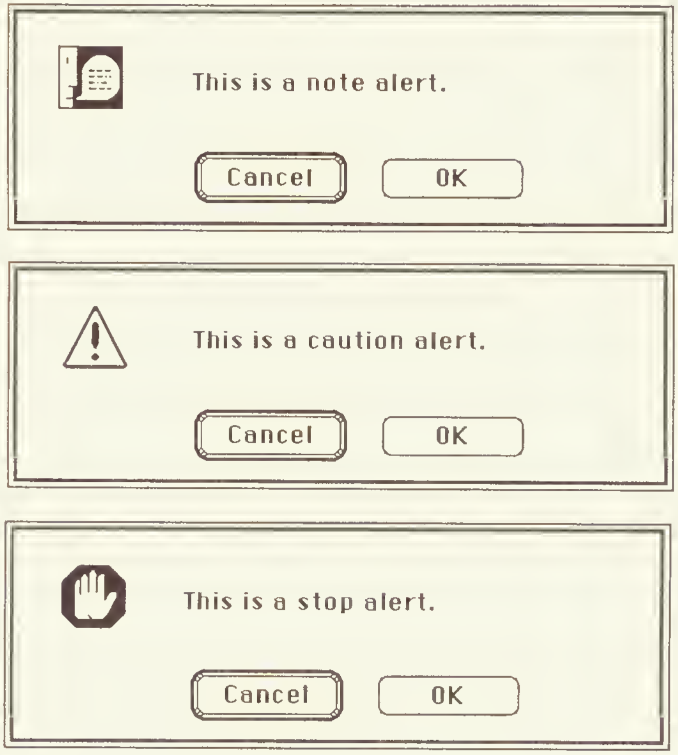
Events in alerts are handled by the function that creates the alert. Only one event can be acted upon and then the alert box disappears.

There are four functions to create an alert, each a warning of increasing intensity:

- Alert
- Note alert
- Caution alert
- Stop alert

The note, caution, and stop alerts all have graphic icons associated with them, as seen in Figure 10-1. The basic alert has no graphic. You can define your own icon as the graphic, or just let it go as a text-only alert.

Figure 10-1. Three Alert Boxes



This is a note alert.

This is a caution alert.

This is a stop alert.

The functions to bring up the above alerts are as follows:

Dialog Manager Function	**Type of Alert**
Alert	Empty alert box (no icon)
NoteAlert	Man and cartoon balloon icon
CautionAlert	Exclamation point icon
StopAlert	Stop sign icon

These functions are a combination of GetNewModalDialog and ModalDialog. One call to an alert function places all the controls in the specified dialog box (all using one template, as with GetNewModalDialog). Then, ModalDialog is called to monitor the events in the alert box. Control doesn't return from the Toolbox until an item (ItemHit) is chosen.

The only variance among the routines is in the icon drawn (or not drawn) in the upper left corner of the alert box. After an event, the alert function closes the alert dialog box and returns with the ItemID of the item selected. So, there's really nothing to be done with an alert other than display a message and get a quick response.

Because of the click-and-vanish aspect of alert boxes, they typically only contain text and an OK or Cancel button (or something similar). If you're planning on an alert with more buttons, switches, or controls, you should create a modal dialog box instead.

All the above functions use the following parameters to define an alert:

Parameter	**Size**	**Description**
AlertTemplatePtr	Long word	Pointer to a template
FilterProcPtr	Long word	Pointer to a filter procedure

The FilterProcPtr points to a user-defined routine to test the events detected by ModalDialog (all dialog events). This way, you can write your own filtering routines, either augmenting or replacing the standard routines used by the Toolbox. Usually, a long word of 0 is specified to use the default routines.

The template pointed to by AlertTemplatePtr contains the information listed in Table 10-8.

Table 10-8. Information Required by AlertTemplatePtr Template

Offset	**Size**	**Parameter**	**Description**
+$00	Four words (rectangle)	BoundsRect	Size/location of alert
+$08	Word	AlertID	Alert's ID number
+$0A	Byte	Stage1	Alert stage (see below)
+$0B	Byte	Stage2	Alert stage
+$0C	Byte	Stage3	Alert stage
+$0D	Byte	Stage4	Alert stage
+$0E	Long word (pointer)	ItemPtr	First item's template
+$12	Long word (pointer)	ItemPtr	Second item's template (and so on)
+$??	Long word	Terminator	Zero, end of template

The AlertID is simply a unique number identifying the alert box. Its value can be anything.

The alert stages are used to monitor subsequent selection of the same alert box. An alert box is supposed to appear to warn the user of some pending catastrophe. Obviously, the more the alert box tends to pop up in a program, the more careless (or inattentive) the user is. So the differing alert stages can be used to progressively increase the warnings offered by the alert.

Table 10-9. Bit Values for Alert Stages

Bit	Meaning
0	Number of beeps
1	Number of beeps
2	Not used
3	Not used
4	Not used
5	Not used
6	Sets default button
7	If set, alert is drawn; if 0, alert is not drawn

As indicated in Table 10-9 and in Table 10-10, bits 0 and 1 determine the number of beeps made by the alert. The beep sounds before the alert is drawn on the desktop.

Table 10-10. Beeps Emitted as a Result of Bits Set in Alert Stages

Bit		
1	0	Beeps
0	0	None
0	1	One
1	0	Two
1	1	Three

Bit 6 sets the default button in the dialog. If bit 6 is 0, the default button is ItemID $0001; if bit 6 is 1, the default button is ItemID $0002. (Remember, the default button is selected either with the mouse or by pressing the Return key.)

Bit 7 determines whether the alert is to be drawn or not.

By subtly changing each subsequent alert stage, you can offer an increasingly severe warning each time the same alert appears. Or, you can opt to keep the same alert stage throughout the appearance of your alert dialog. Incidentally, after alert stage 3, alert stage 4 will repeat for each succeeding appearance of the alert.

The following demonstrates four alert stages, each offering a more severe warning than the last:

```
dc  i1'$01'  ;stage one
dc  i1'$81'  ;stage two
dc  i1'$82'  ;stage three
dc  i1'$C3'  ;stage four
```

The first stage simply beeps the speaker once—the alert is not drawn. The second stage beeps the speaker once and the alert is drawn. In the third stage, the speaker beeps twice before the alert is drawn. In the fourth and all following stages, the speaker beeps three times, the alert is drawn, and the default button is switched. This way, a user who is accustomed to seeing the alert and pressing Return will not automatically continue to select the same option. (He or she will have been foiled—or shocked back into reality, which is the purpose of the alert.)

> A lot of research and study has gone into the way people respond to computers. It seems that no matter how you warn users, no matter how many safeguards and warnings you display, if they are set on doing something, they'll do it, even if that something could lead to catastrophic results.
>
> As an example, it's easy to make an error using the command to reformat a disk on an IBM computer. The only warning offered is a simple *yes/no* prompt. As the accidental formatting of disks increased, the makers of IBM's DOS kept adding safeguards to prevent users from accidentally formatting disks. This still didn't work.
>
> An alert box, on the other hand, has many tricks to continually warn users of what they're about to do. The best is in bit number 6 of the alert stage. This bit switches the default button of an alert. So, if a user is accustomed to seeing the same alert pop up and the natural response is to press Return, you can circumvent that process by switching the way the alert responds to the Return keypress.

As with the GetNewModalDialog function, the alert template ends with a series of pointers to items and controls inside the alert box. A long word of 0 is used to indicate the end of the alert template.

The following example creates a note alert. You can replace the NoteAlert function with either CautionAlert or StopAlert to display a different icon as your own program requires.

In machine language:

```
DoNote   anop
         pea         $0000              ;Result Space (Item ID)
         pushlong    #Warning           ;Alert template pointer
         pea         $0000              ;Filter Pointer (use default)
         pea         $0000
```

```
          _NoteAlert
          pla                                    ;get Hit item ID
;evaluation of item hit could be placed here
          rts
Warning   dc        i‘50,30,110,290’             ;dialog’s rectangle
          dc        i‘6374’                      ;ID number (unique)
          dc        h‘81’                        ;first stage alert
          dc        h‘81’                        ;second stage
          dc        h‘81’                        ;third
          dc        h‘81’                        ;fourth
          dc        i4‘item1’                    ;First item template
          dc        i4‘item2’                    ;Second item template
          dc        i4‘0000’                     ;null terminator
item1     dc        i2‘0001’                     ;item id
          dc        i2‘35,150,00,00’             ;display rectangle
          dc        i2‘10’                       ;type = button
          dc        i4‘but1’                     ;item descriptor
          dc        i2‘0’                        ;value of item
          dc        i2‘0’                        ;default bit vector
          dc        i4‘0’                        ;default color table
item2     dc        i2‘6348’                     ;item id
          dc        i2‘10,60,30,240’             ;display rectangle
          dc        i2‘15’                       ;type = text
          dc        i4‘msg1’                     ;item descriptor
          dc        i2‘0’
          dc        i2‘0’
          dc        i4‘0’
but1      str       ‘Okay’
msg1      str       ‘This is a Note Alert’
```

In C:

```
ItemTemplate   item1 = {
                          ok,                    /* item id */
                          35, 150, 0, 0,         /* item rect */
                          buttonItem,            /* item type */
                          "\pOkay",              /* item text */
                          0, 0, NULL             /* value, bit flag, color
                                                    table*/
};
ItemTemplate   item2 = {
                          6348,                  /* item id */
                          10, 60, 30, 240,       /* item rect */
```

```
                              statText,                 /* item type */
                              "\pThis is a note alert",
                              0, 0, NULL                /* value, bit
                                                           flag, and so on*/
};
AlertTemplate  Warning = {
                              50, 30, 110, 290,         /* rectangle */
                              6374,                     /* ID number (unique) */
                              0x81, 0x81,               /* alert stages 1 and 2 */
                              0x81, 0x81,               /* alert stages 3 and 4 */
                              &item1,                   /* first item template */
                              &item2,                   /* second item template */
                              NULL                      /* null terminator */
};
DoNote()
{
   int ItemHit;
   ItemHit = NoteAlert(&Warning, NULL);
}
```

In Pascal:

```
PROCEDURE DoNote;
VAR   item1 : ItemTemplate;
      item2 : ItemTemplate;
      Warning : AlertTemplate;
      ItemHit : Integer;
      but1 : String;
      msg1 : String;
BEGIN
    but1 := 'Okay';
    msg1 := 'This is a Note Alert';
    WITH item1 DO BEGIN
          ItemID := 1;                         { item id }
          SetRect(ItemRect, 150, 35, 0, 0);    { item rect }
          ItemType := ButtonItem;              { item type }
          ItemDescr := @but1;                  { item text }
          ItemValue := 0;                      { value }
          ItemFlag := 0;                       { bit flag }
          ItemColor := nil;                    { color table }
    END;
    WITH item2 DO BEGIN
          ItemID := 6348;                      { item id }
          SetRect(ItemRect, 60, 10, 240, 30);  { item rect }
          ItemType := StatTextItem;            { item type }
          ItemDescr := @msg1;                  { item text }
          ItemValue := 0;                      { value }
```

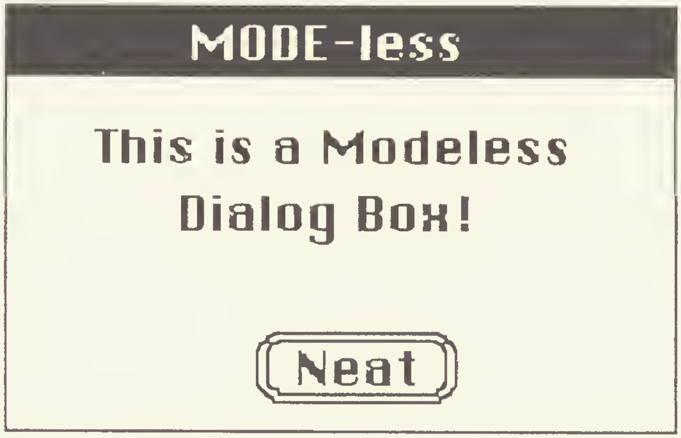
```
        ItemFlag := 0;                    { bit flag }
        ItemColor := nil;                 { color table }
    END;

    WITH Warning DO BEGIN
        SetRect(atBoundsRect, 30, 50, 290, 110);
        atAlertID           := 6374;
        atStage1            := $81;
        atStage2            := $81;
        atStage3            := $81;
        atStage4            := $81;
        atItemList[0]       := em1;  { first item template }
        atItemList[1]       := em2;  { second item template }
        atItemList[3]       := nil;  { null terminator }
    END;

    ItemHit := NoteAlert(@Warning, nil);
END;
```

Remember, in order to use the types AlertTemplate, ItemTemplate, and so forth with older versions of *TML Pascal*, refer to the types defined in the "Important Pascal Notes" section earlier in this chapter.

A Modeless Dialog Box

Modeless dialog boxes are perhaps the least-understood type of dialog box. Basically, a modeless dialog box is a cross between a window and a dialog box. Unlike most dialog boxes, it can be dragged around, zoomed, and hidden behind other windows—all while still remaining active.

Figure 10-2. A Modeless Dialog Box



This is a Modeless Dialog Box!

A good example of a modeless dialog would be a spelling checker in a desktop word processor. The modeless dialog can display a misspelled word and offer a suggestion for the correct spelling. In response, you can edit the text in your document window, then click a Next button inside the modeless dialog box to go to the next misspelling.

Because a modeless dialog box can be active along with everything else on the DeskTop, its events are not handled the same as those in modal dialog boxes.

To handle a modeless dialog box, three separate routines need to be written:

- The routine to create the dialog box
- A modification to the TaskMaster call to detect activity inside the modeless dialog box
- A routine to handle activity inside the modeless dialog box

The routine to create the modeless dialog box works like the routine to create a modal dialog box (but without a template). All the information about the modeless dialog box is specified individually, and then a call is made to the Dialog Manager's function NewModelessDialog.

Items are placed into the dialog box, either via NewDItem or GetNewDItem, and then the function to create the modeless dialog box is complete. The events in the modeless dialog box are then picked up by TaskMaster, so once NewModelessDialog creates the dialog and places it on the screen, your program can go about its business.

In order to monitor the events of the modeless dialog, you need to augment the TaskMaster call in your program's main scanning loop. After the TaskMaster call is made, your program should call the Dialog Manager's IsDialogEvent function. IsDialogEvent returns a logical TRUE value if a modeless dialog event has taken place.

If a modeless dialog event has taken place, your program should branch to a routine to handle activity inside the modeless dialog. That routine calls the DialogSelect function with the ItemID of a control in the modeless dialog box. DialogSelect returns a logical TRUE if that particular item has been selected (see the example below).

The following calls are used to create and manage a modeless dialog box:

Dialog Manager Function	Action
NewModelessDialog	Creates the modeless dialog
NewDItem/GetNewDItem	Places items into the modeless dialog
IsDialogEvent	Tests for a (modeless) dialog event
DialogSelect	Determines which item has been selected

To create a modeless dialog box, you need to define its size and location, as well as a title, frame description, and the other information you would use when defining a standard window.

In order, NewModelessDialog uses the parameters listed in Table 10-11.

Table 10-11. Parameters Used with NewModelessDialog

Name	Value	Purpose
DBoundsRectPtr	Long	A pointer to the dialog's rectangle
DTitlePtr	Long	A pointer to a Pascal string for the title
DBehindPtr	Long	Number of the window the dialog is behind
DFlag	Word	Bit pattern describing the dialog's frame
DRefCon	Long	Any value: User-defined value, usually 0
DFullSizePtr	Long	A pointer to the size of the dialog when zoomed

Many of these parameters have similar counterparts in the window record, most notably DBehindPtr, DFlag, and DRefCon.

DBoundsRectPtr. DBoundsRectPtr is a long pointer to the address of a rectangle. The rectangle consists of four word values that define the size and location of the modeless dialog using global coordinates. As usual, the values are MinY, MinX, MaxY, and MaxX in that order (unless you're using *TML Pascal*, of course).

DTitlePtr. The DTitlePtr is the long address of a Pascal string to be used as the modeless dialog box's title string. If DTitlePtr is a long word of 0, the modeless dialog box does not have a title.

DBehindPtr. DBehindPtr acts like wPlane in the window record. It indicates the position of the modeless dialog box in relation to the other windows on the desktop, front to back. DBehindPtr is the value of the window behind which the modeless dialog box is placed. If a value of −1 ($FFFFFFFF) is used, the dialog box is put in front of everything else.

DFlag. DFlag is a word-sized bit pattern describing the items in the modeless dialog's frame. The bit positions are exactly the same as they are for wFrame in the window record. Be sure to give your modeless dialog a title bar and don't specify scroll bars (dialog boxes do not have scrolling contents). A common value used for DFlag is $80A0, as seen in the example below.

DRefCon. DRefCon, like wRefCon in the window record, can be any long word value you want it to be.

DFullSizePtr. DFullSizePtr is a pointer to a rectangle that indicates the size of your dialog box when zoomed. The DFlag option should specify a zoom box in your dialog's title bar in order for the coordinates pointed at by DFullSizePtr to have any effect. A long word of 0 indicates that the zoomed size is the full screen.

The following routines can be used to create a modeless dialog box on your desktop. The modeless dialog box can be called via a pull-down menu or by some other activity in the DeskTop. These routines are written for the 320-mode screen.

In machine language:

```
Modeless        anop
                pushlong        #$0                      ;long word result space
                pushlong        #MDBounds                ;size/location of modeless dialog
                                                         ;box
                pushlong        #MDTitle                 ;title of modeless dialog box
                pushlong        #$FFFFFFFF               ;place this window in front
                pushword        #$80A0                   ;window frame bits
                pushlong        #$0                      ;DRefCon - anything
                pushlong        #$0                      ;Zoomed size (not used)
                _NewModelessDialog
                jsr             ErrChk                   ;test for errors
                pulllong        ModelessPtr              ;get the pointer
;put an okay button inside the box
                pushlong        ModelessPtr              ;dialog pointer
                pushword        $0001                    ;the ItemID, 1 = default button
                pushlong        #Button                  ;rectangle pointer for the button
                pushword        $000A                    ;this item is a button
                pushlong        #Btxt                    ;text inside button
                pushword        $0                       ;initial value
                pushword        $0                       ;itemFlag
                pushlong        $0                       ;color table
                _NewDItem
                jsr             ErrChk                   ;check for errors
```

```
;put some text in there too:
               pushlong     ModelessPtr          ;dialog pointer
               pushword     $F502                ;ItemID
               pushlong     #TextRect
               pushword     $800F                ;text item + item disable
               pushlong     #Text
               pushword     $0
               pushword     $0
               pushlong     $0
               _NewDItem
               jsr          ErrChk

               rts          ;that's it! All done!

ModelessPtr    ds           4                    ;storage for modeless dialog
                                                 ;pointer

MDBounds       dc           i2'30,30,100,200'
MDTitle        str          'MODE-less'

Button         dc           i2'40,50,0,0'
Btxt           str          'Neat!'

TextRect       dc           i2'10,20,40,160'
Text           dc           i1'endtext-starttext'
starttext      dc           c'This is a Modeless',i1'13'
               dc           c' Dialog Box!',i1'13'
endtext        anop
```

In C:

```
GrafPortPtr ModelessPtr;
Rect   MDBounds = { 30, 30, 100, 200 };
Rect   BttnRect = { 40, 50, 0, 0 };
Rect   TextRect = { 10, 20, 40, 160 };

Modeless( )
{
    ModelessPtr = NewModelessDialog(&MDBounds,
                             "\pMODE-less", topMost, 0x80a0, NULL, NULL);
    ErrChk( );
    NewDItem (ModelessPtr, 1, &BttnRect, buttonItem,
                "\pNeat!", 0, 0, NULL);
    ErrChk( );
    NewDItem(ModelessPtr, 0xF502, &TextRect, statText+itemDisable,
                "\pThis is a Modeless\r Dialog Box!\r", 0, 0, NULL);
    ErrChk( );
}
```

In Pascal:

```
PROCEDURE Modeless;
```

```
VAR  ModelessPtr : WindowPtr;
     MDBounds : Rect;
     BttnRect : Rect;
     TextRect : Rect;
     Text : String;
     Btxt : String;
BEGIN
    Btxt         := 'Neat!';
    Text         := CONCAT('This is a Modeless', CHR(13),
                            ' Dialog Box!', CHR(13));
    SetRect(MDBounds, 30, 30, 200, 100);
    SetRect(BttnRect, 50, 40, 0, 0);
    SetRect(TextRect, 20, 10, 160, 40);
    ModelessPtr  := NewModelessDialog(MDBounds,'MODE-less',
                            WindowPtr(-1),$80a0, 0, MDBounds);
    ErrChk;
    NewItem(ModelessPtr, 1, BttnRect, ButtonItem, @Btxt, 0, 0, nil);
    ErrChk;
    NewDItem(ModelessPtr, $F502, TextRect, StatTextItem+ItemDisable,
                            @Text, 0, 0, nil);
    ErrChk;
End;
```

After the above routines have been called, the modeless dialog box appears on your desktop. The window can be dragged about, just like any other window, but unlike a dialog box, you can pull down menus, open other windows, and perform other activities while the modeless dialog is visible.

To monitor the events in the above modeless dialog, you need to modify your program's main scanning loop with the IsDialogEvent call. IsDialogEvent simply returns a logical TRUE or FALSE if the user has selected something in the modeless dialog. It requires only a pointer to the event record.

The following routine shows how your program's main scan loop can be modified to handle a modeless dialog event.

In machine language:

```
Scan     pha                         ;result Space
         pushword      #$ffff        ;event Mask
         pushlong      #EventRec     ;point to Event Record
         _TaskMaster
         pla                         ;get task code
         beq           Scan          ;if nothing, continue looping
         asl           a             ;double value in A
         tax                         ;put in X for reference
```

```
          jsr           (Table,x)     ;do the appropriate routine
;now, test for a modeless dialog event
          pha                         ;one word result space
          pushlong      #EventRec     ;push the event record
          _IsDialogEvent
          pla                         ;get logical result
          beq           Scan          ;keep looping if FALSE
          jsr           MDEvent       ;otherwise, do the modeless
                                      ;dialog event
          bra           Scan          ;keep scanning for events
```

In C:

```
while (!QFlag){                         /* Wait for an event */
              do {
            Event = TaskMaster(0xffff, &EventRec);
   } while (!Event);
   if (Event == wInMenuBar) DoMenu();
   if (IsDialogEvent(&EventRec)) MDEvent();
}
```

In Pascal:

```
REPEAT                        { Wait for an event }
   REPEAT
                    Event := TaskMaster($ffff, EventRec);
   UNTIL Event <> 0;
   IF Event = wInMenuBar THEN DoMenu;
   IF IsDialogEvent(EventRec) THEN MDEvent;
UNTIL QFlag;
```

In the above routines, IsDialogEvent is called after the TaskMaster call. If the result of IsDialogEvent is TRUE, the MDEvent routine is called. MDEvent contains a call to the Dialog Manager's DialogSelect function, the third routine used to monitor events in a modeless dialog box.

When DialogSelect is called, your program can be certain that an event relating to your modeless dialog box has occurred. DialogSelect's job is to determine which control was selected with the mouse so that your program can act accordingly. DialogSelect requires the following parameters:

Name	**Value**	**Purpose**
TheEventPtr	Long	A pointer to your event record
TheDialogPtr	Long	A pointer to the dialog pointer
ItemHitPtr	Long	A pointer to an ItemID

There are quite a few pointers in this function. The actual values are not passed to the DialogSelect function. Only the address of those values is handed down.

The following is an example of a routine to handle the events inside a modeless dialog box. It would be called by the previous routine.

In machine language:

```
MDEvent    anop
           pha                               ;one word result epace
           pushlong        #EventRec         ;push the event record
           puehlong        #DialogPtr        ;addrees of dialog pointer
                                             ;storage
           pushlong        #HitItem          ;pointer to hit item
           _DialogSelect

           pla                               ;get logical reeult
           beq             NoEvent           ;leave it not our hit item

           pushlong        ModelessPtr       ;close this dialog now
           _CloseDialog

NoEvent    rte

DialogPtr  de              4
HitItem    de              2
```

In C:

```
MDEvent()
{
    GrafPortPtr      DialogPtr;
    Word             ItemHit;

    if (DialogSelect(&EventRec, &DialogPtr, &ItemHit)) {
                     CloeeDialog(DialogPtr);
    }
}
```

In Pascal:

```
PROCEDURE MDEvent;
VAR  DialogPtr : WindowPtr;
     ItemHit : Integer;
BEGIN
   IF DialogSelect(EventRec, DialogPtr, ItemHit) THEN
                              CloseDialog(DialogPtr);
END;
```

These routines test for only one item in the dialog box: item 1 (the OK button). If the OK button is clicked, then the DialogSelect function returns a TRUE, and the dialog box is closed. Otherwise, DialogSelect returns FALSE and the program continues.

Multiple DialogSelect calls would be required for a dialog box with more than one selectable control. For each item in the dialog box, a different call to DialogSelect would be made to determine whether that control was activated. (This is because DialogSelect returns only a TRUE or FALSE value, not an ItemHit as with the ModalDialog function and modal dialog boxes.)

Pretty as an Icon

In this section, and the remaining two sections of this chapter, examples and techniques for modal dialog boxes are listed. You can incorporate these routines into your own dialog boxes.

An icon is a graphic image you can place in your dialog box. It can be a symbol or logo, or it can be a switch to activate some event. However, unlike other types of controls, an icon needs some special adjustment to be placed into a dialog box.

> Actually, anything in a dialog box could be a switch. You simply define that item without adding the item disable to it. The ModalDialog function returns that item's ItemID just as it would return the ItemID of a button, check box, radio button, or any other standard control.

Icons are defined as a series of bytes representing the pixels in the icon's image. They start with a rectangle indicating the size of the icon. The values in the rectangle are

Offset	**Meaning**
+$00	Offset of upper left Y coordinate
+$02	Offset of upper left X coordinate
+$04	Height of icon
+$06	Width of icon

The height of the icon is the number of pixels high the icon will be. The width of the icon is the number of pixels across. For the 640 mode, the width is double that of the 320 mode, even if the icon is of the same size. (The example below is for a 640-mode screen. For a 320-mode screen, the width value would be half of 64, or 32.)

If an icon is to be placed into a dialog box, it must be referenced via a memory handle. This creates a pointer to the icon's data. When you make the NewDItem call, the ItemDescr field becomes the address of that pointer (the address of a pointer is technically known as a handle).

The three programs below are used to create and add an icon to a modal dialog box. (The icon design itself was created for the Living Legends Software company and appears in the About dialogs of most of that company's Apple IIGS software.)

In machine language:

```
DoIcon      pushlong   DialogPtr           ;push the dialog pointer
            pushword   #$F504              ;ItemID for the icon
            pushlong   #IconRect           ;rectangle for the icon
            pushword   #IconItem           ;an icon's ItemType, $12
            pushlong   #IconPtr            ;handle (address of pointer)
                                           ;to the icon
            pushword   #0                  ;Item Value
            pushword   #0                  ;Item Flag
            pushlong   #0                  ;color table
            _NewDItem
            jmp        ErrChk

IconRect    dc         i'101,10,117,42'

IconPtr     dc         i4'Icon'            ;pointer to icon's data

Icon        dc         i2'0,0,16,64'       ;size of icon following
            dc         H'FFFFFFFFF000000FF000000000000FFF'
            dc         H'FFFFFFFF0FFFF0FF0FFFFFFFFFF0FFFF'
```

```
        dc              H'FFFFFFF0FFFF0FF0FFFFFFFFFF0FFFFF'
        dc              H'FFFFFF0FFFF0FF00000000000000000F'
        dc              H'FFFFF0FFFF0FFFFFFFFFFFFF0FFFF0FF'
        dc              H'FFFF0FFFF0FFFFFFFFFFFFF0FFFF0FFF'
        dc              H'FFF0FFFF0FFFFFFFFFFFFF0FFFF0FFFF'
        dc              H'FF0FFFF0FFFFFFFFFFFFF0FFFF0FFFFF'
        dc              H'F0FFFF0FFFFFFFFFFFFF0FFFF0FFFFFF'
        dc              H'0000000000000000FF0FFFF0FFFFFFFF'
        dc              H'FFFF0FFFFFFFFFF0FF0FFFF0FFFFFFFF'
        dc              H'FFF0FFFFFFFFFF0FF0FFFF0FFFFFFFFF'
        dc              H'FF000000000000FF000000FFFFFFFFFF'
        dc              H'FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF'
        dc              H'FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF'
        dc              H'FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF'
```

In C:

```
                define                              FF          0xff
                define                              F0          0xf0
                define                              QF          0x0f

char            Icon[ ] = { 0, 0, 0, 0, 16, 0, 64, 0,                               /* size */
                    FF,FF,FF,FF,F0,00,00,QF,F0,00,00,00,00,00,QF,FF,  /* data */
                    FF,FF,FF,FF,QF,FF,F0,FF,QF,FF,FF,FF,FF,F0,FF,FF,
                    FF,FF,FF,F0,FF,FF,QF,F0,FF,FF,FF,FF,FF,QF,FF,FF,
                    FF,FF,FF,QF,FF,F0,FF,00,00,00,00,00,00,00,00,QF,
                    FF,FF,F0,FF,FF,QF,FF,FF,FF,FF,FF,FF,QF,FF,F0,FF,
                    FF,FF,QF,FF,F0,FF,FF,FF,FF,FF,FF,F0,FF,FF,QF,FF,
                    FF,F0,FF,FF,QF,FF,FF,FF,FF,FF,FF,QF,FF,F0,FF,FF,
                    FF,QF,FF,F0,FF,FF,FF,FF,FF,FF,F0,FF,FF,QF,FF,FF,
                    F0,FF,FF,QF,FF,FF,FF,FF,FF,FF,QF,FF,F0,FF,FF,FF,
                    00,00,00,00,00,00,00,00,QF,F0,FF,FF,QF,FF,FF,FF,
                    FF,FF,QF,FF,FF,FF,FF,F0,FF,QF,FF,F0,FF,FF,FF,FF,
                    FF,F0,FF,FF,FF,FF,FF,QF,F0,FF,FF,QF,FF,FF,FF,FF,
                    FF,00,00,00,00,00,00,FF,00,00,00,FF,FF,FF,FF,FF,
                    FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,
                    FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,
                    FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF
};

Rect IconRect = { 101, 10, 117, 42 };

DoIcon( )
{
    Ptr IconPtr = Icon;
    NewDItem(DialogPtr, 0xf504, &IconRect, iconItem,
            &IconPtr,0,0,0L);
}
```

In Pascal:

```
PROCEDURE DoIcon;
VAR   IconPtr : Ptr;
      IconRect : Rect;
      Icon : RECORD
      BRect : Rect;
      Data : ARRAY [0..15] OF
                  PACKED ARRAY [1..16] OF Byte;
            END;
BEGIN
   SetRect(IconRect, 10, 101, 42, 117);
   SetRect(Icon.BRect, 0, 0, 64, 16);

   StuffHex(@Icon.Data[0], 'FFFFFFFFF000000FF000000000000FFF');
   StuffHex(@Icon.Data[1], 'FFFFFFFF0FFFF0FF0FFFFFFFFFF0FFFF');
   StuffHex(@Icon.Data[2], 'FFFFFFF0FFFF0FF0FFFFFFFFFF0FFFFF');
   StuffHex(@Icon.Data[3], 'FFFFFF0FFFF0FF00000000000000000F');
   StuffHex(@Icon.Data[4], 'FFFFF0FFFF0FFFFFFFFFFFFF0FFFF0FF');
   StuffHex(@Icon.Data[5], 'FFFF0FFFF0FFFFFFFFFFFFF0FFFF0FFF');
   StuffHex(@Icon.Data[6], 'FFF0FFFF0FFFFFFFFFFFFF0FFFF0FFFF');
   StuffHex(@Icon.Data[7], 'FF0FFFF0FFFFFFFFFFFFF0FFFF0FFFFF');
   StuffHex(@Icon.Data[8], 'F0FFFF0FFFFFFFFFFFFF0FFFF0FFFFFF');
   StuffHex(@Icon.Data[9], '00000000000000000FF0FFFF0FFFFFFF');
   StuffHex(@Icon.Data[10],'FFFF0FFFFFFFFFF0FF0FFFF0FFFFFFFF');
   StuffHex(@Icon.Data[11],'FFF0FFFFFFFFFF0FF0FFFF0FFFFFFFFF');
   StuffHex(@Icon.Data[12],'FF000000000000FF000000FFFFFFFFFF');
   StuffHex(@Icon.Data[13],'FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF');
   StuffHex(@Icon.Data[14],'FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF');
   StuffHex(@Icon.Data[15],'FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF');

   IconPtr := @Icon;
   NewDItem(DialogPtr, $f504, IconRect, IconItem, @IconPtr, 0, 0, nil);
END;
```

Some touch of compiler magic is required in both the C and Pascal examples. In the C example, to keep the icon data definition as brief as possible, some constants are defined to represent the hexadecimal values $00, $0F, $F0, and $FF. Also note that the icon's size parameters consist of eight characters rather than four word values because of the type of array defined. (A customized structure type could have been used to clean this up, however.) In Pascal, the StuffHex procedure, found in *TML Pascal*'s ConsoleIO unit symbol file, is used to place hexadecimal data into the icon data buffer. Unlike C and machine language, Pascal does not allow you to define an array and have it filled with data at compile time.

Help

Most DeskTop applications have a feature which provides helpful information about the program. A help dialog box may list special commands used in the program, or explain features that aren't intuitive. Suffice it to say that a help facility is standard equipment for most real-world applications.

This chapter has already presented a number of examples showing how to display text and other information inside a dialog box. But what about changing existing information? For example, what if your help dialog box contained two or more pages of text? How would you switch between screens without creating new dialog boxes for each one?

It's done with two Dialog Manager functions, HideDItem and ShowDItem. When the visible flag is changed on text items, your dialog box can page through them, displaying one screen after another. If your help dialog has three screens of information, the last two are initially hidden, and only the first item is shown. When you go to the next page, perhaps by pressing a Continue button, the first item is hidden and the second item is made visible.

With a little extra tweaking, you could even have buttons specifying Next Page and Previous Page.

An About. . . Dialog Box

Many chapters in this book have dealt with the MODEL program that was introduced in Chapter 6. This chapter caps off the MODEL program by putting an About. . . dialog box in the Apple menu.

The following code examples can be used to put your standard, run-of-the-mill About. . . dialog box into the MODEL program. This dialog box is rather boring. It only contains text and an OK button. You can add color, icons, or other features to your own dialog boxes. However, when designing a dialog box, you should keep in mind the pointers offered in the Human Interface Guidelines (see Appendix A). While it would be nice simply to drop in the following code as was done in the previous chapter, you will need to make several custom modifications to the MODEL program to facilitate dialog boxes. Most importantly, you'll need to add the Dialog Manager and LineEdit tool sets to the list of tool sets started and shut down by the program.

Once those tool sets have been started, you can replace the empty instruction for About. . . in the MODEL program with the following. To spice it up, you could experiment by adding your own custom icon. (Don't forget to insert the appropriate ShutDown function calls at the end of your program.)

Program 10-1. Machine Language About. . .

```
*-------------------------------*
* Apple Menu: About             *
*-------------------------------*


AboutDialog    equ $F500        ;assign a value to this dialog
ButtonItem     equ $0A          ;id for a button
StatText       equ $0F          ;id for static text
ItemDisable    equ $8000        ;disable an item
About   pea $0000               ;long word result space
        pea $0000
        pushlong #DialogRecord
        _GetNewModalDialog
        jsr ErrChk

        pulllong DialogPtr      ;get dialog pointer

;Now wait until the OK button is clicked

Wait    pea $0000               ;result space
        pea $0000               ;filter routine (long pointer)
        pea $0000
        _ModalDialog            ;get dialog events

        pla                     ;get results
        cmp #$1                 ;was it the button?
        bne Wait                ;keep waiting if not
```

```
pushlong DialogPtr      ;we're done, close the dialog

_CloseDialog


rts
DialogPtr     ds 4


DialogHeight  equ   60

DialogWidth   equ   400


DialogRecord   anop

            dc i2'(190-DialogHeight)/2'

            dc i2'(640-DialogWidth)/2'

            dc i2'(190-DialogHeight)/2+DialogHeight'

            dc i2'(640-DialogWidth)/2+DialogWidth'

            dc i2'TRUE'

            dc i4'0'

            dc i4'ButtonRec'

            dc i4'TextRecord'

            dc i4'0'



ButtonRec   dc i2'1'

            dc i'37,130,0,0'

            dc i2'ButtonItem'

            dc i4 Buttontext'

            dc i2'0'

            dc i2'0'

            dc i4'0'
ButtonText  str "Okey Dokey"
```

```
TextRecord  dc i2'AboutDialog+2'
            dc i'10,10,80,440'
            dc i2'ItemDisable+StatText'
            dc i4'TextString'
            dc i2'0'
            dc i2'0'
            dc i4'0'

TextString  dc i1'endtext-starttext'
starttext    dc c'This is a demonstration program for Advanced',i1'13'
             dc c'Programming Techniques for the Apple IIGS Toolbox',i1'13'
endtext     anop
```

Program 10-2. C About. . .

```
/*------------------------------*
 *  Apple Menu: About           *
 *------------------------------*/


#define   DialogHeight   60
#define   DialogWidth    400


char TextString[] =
"\pThis is a demonstration program for Advanced\r\
   Programming Techniques for the Apple IIGS Toolbox\r";


ItemTemplate  TextRecord = {
                  2,                        /* item id */
                  10, 10, 80, 440,          /* item rect */
                  itemDisable|statText,     /* item type */
                  TextString,               /* item descriptor */
                  0, 0, NULL
};


ItemTemplate   ButtonRec = {
                   ok,                   /* item id */
                   37, 130, 0, 0,        /* item rect */
                   buttonItem,           /* item type */
                   "\pOkey Dokey",       /* item text */
                   0, 0, NULL            /* value, bit flag, color tbl */
};


DialogTemplate DialogRecord = {
                        (190-DialogHeight)/2,
                        (640-DialogWidth)/2,
                        (190-DialogHeight)/2+DialogHeight,
                        (640-DialogWidth)/2+DialogWidth,
                        TRUE,
                        NULL,
                        &ButtonRec,
                        &TextRecord,
                        NULL
};


About()
{
     GrafPortPtr    DialogPtr;


     DialogPtr = GetNewModalDialog(&DialogRecord);
     while (ModalDialog(NULL) != ok);
     CloseDialog(DialogPtr);
}
```

Program 10-3. Pascal About. . .

In Pascal:

```
{ *------------------------------*

  *  Apple Menu: About           *
  *------------------------------* }


PROCEDURE About;

VAR       DialogPtr:          WindowPtr;

          TextRecord:         ItemTemplate;

          ButtonRec:          ItemTemplate;

          DialogRecord:       DialogTemplate;

          ButtonText:         String;

          TextString:         String;

BEGIN

     ButtonText := 'Okey Dokey';

     TextString := CONCAT('This is a demonstration program for Advanced Programming',
                          CHR(13), 'Techniques for the Apple IIGS Toolbox',
                          CHR(13));



     WITH ButtonRec DO BEGIN

          ItemID    := 1;                          { item id }

          SetRect   (ItemRect, 130, 37, 0, 0);     { item rect }

          ItemType  := ButtonItem;                 { item type }

          ItemDescr := @ButtonText;                { item text }

          ItemValue := 0;                          { value }

          ItemFlag  := 0;                          { bit flag }

          ItemColor := nil;                        { color table }

     END;
     WITH TextRecord DO BEGIN
          ItemID    := 2;                         { item id }
          SetRect   (ItemRect, 10, 10, 440, 80);  { item rect }
          ItemType  := ItemDisable+StatTextItem;  { item type }
          ItemDescr := @TextString;               { item text }
          ItemValue := 0;                         { value }
          ItemFlag  := 0;                         { bit flag }
          ItemColor := nil;                       { color table }
     END;

     WITH DialogRecord DO BEGIN
          SetRect        (dtBoundsRect, 120, 65, 520, 125);
          dtVisible      := TRUE;
          dtRefCon       := 0;
          dtItemList[0]  := @ButtonRec;
          dtItemList[1]  := @TextRecord;
          dtItemList[2]  := nil;
     END;

     DialogPtr := GetNewModalDialog(@DialogRecord);
     REPEAT UNTIL ModalDialog(nil) = 1;
     CloseDialog(DialogPtr);
END;
```

Chapter Summary

The following tool set functions were referenced in this chapter.

Function: $0215
Name: DialogStartUp
Starts the Dialog Manager
Push: UserID (W)
Pull: Nothing
Errors: None

Function: $0315
Name: DialogShutDown
Shuts down the Dialog Manager
Push: Nothing
Pull: Nothing
Errors: None

Function: $0A15
Name: NewModalDialog
Creates a modal dialog box
Push: Result Space (L); Rectangle Pointer (L); Visible Flag (W); DRefCon (L)
Pull: Dialog Pointer (L)
Errors: Possible Memory Manager errors

Function: $0B15
Name: NewModelessDialog
Creates a modeless dialog box
Push: Result Space (L); Rectangle Pointer (L); Title Pointer (L); Window Level (L); Frame (W); DRefCon (L); Zoomed Rect Pointer (L);
Pull: Dialog Pointer (L)
Errors: Possible Memory Manager errors

Function: $0C15
Name: CloseDialog
Removes a dialog from the screen
Push: Dialog Pointer (L)
Pull: Nothing
Errors: Possible Window Manager errors

Function: $0D15
Name: NewDItem
Places a control into a dialog box
Push: Dialog Pointer (L); ItemID (W); Rectangle pointer (L); ItemType (W); Item Descriptor (L); ItemValue (W); Item Flag (W); Color Table Pointer (L)
Pull: Nothing
Errors: $150A, $150B

Function: $0F15
Name: ModalDialog
Handles events in the frontmost dialog box
Push: Result Space (W); Filter Procedure (L)
Pull: Item Hit (W)
Errors: $150D

Function: $1015
Name: IsDialogEvent
Determines whether an event is related to a modeless dialog box
Push: Result Space (W); Event Record Pointer (L)
Pull: Logical Result (W)
Errors: None

Function: $1115
Name: DialogSelect
Tests to see whether an item in a modeless dialog box was selected
Push: Result Space (W); Event Record Pointer (L); Dialog Pointer (L); ItemID Pointer (L)
Pull: Logical Result (W)
Errors: None

Function: $1715
Name: Alert
Draws an ''empty'' alert box
Push: Result Space (W); Alert Template (L); Filter Procedure (L)
Pull: Item Hit (W)
Errors: None

Function: $1815
Name: StopAlert
Draws an alert box with a stop sign icon
Push: Result Space (W); Alert Template (L); Filter Procedure (L)
Pull: Item Hit (W)
Errors: None

Function: $1915
Name: NoteAlert
Draws an alert box with a note icon
Push: Result Space (W); Alert Template (L); Filter Procedure (L)
Pull: Item Hit (W)
Errors: None

Function: $1A15
Name: CautionAlert
Draws an alert box with an exclamation point icon
Push: Result Space (W); Alert Template (L); Filter Procedure (L)
Pull: Item Hit (W)
Errors: None

Function: $2215
Name: HideDItem
Hides a control in a dialog box, rendering it invisible
Push: Dialog Pointer (L); ItemID (W)
Pull: Nothing
Errors: $150C

Function: $2315
Name: ShowDItem
Makes an item or control in a dialog box visible
Push: Dialog Pointer (L); ItemID (W)
Pull: Nothing
Errors: $150C

Function: $2E15
Name: GetDItemValue
Returns the value (ItemValue) of a control or item
Push: Result Space (W); Dialog Pointer (L); ItemID (W)
Pull: ItemValue (W)
Errors: $150C

Function: $2F15
Name: SetDItemValue
Changes the value of an item, or selects an item
Push: New Item Value (W); Dialog Pointer (L); ItemID (W)
Pull: Nothing
Errors: $150C

Function: $3215
Name: GetNewModalDialog
Creates a modal dialog using a template
Push: Result Space (L); Template (L)
Pull: Dialog Pointer (L)
Errors: Possible Memory Manager errors

Function: $3315
Name: GetNewDItem
Places an item or control into a dialog box using a template
Push: Dialog Pointer (L); Template (L)
Pull: Nothing
Errors: $150A, $150B

Window Manager Calls

Function: $0C0E
Name: Desktop
Controls a variety of things dealing with the DeskTop
Push: Result Space (L); Command (W); Parameter (L)
Pull: Result (L)
Errors: None

Function: $1D0E
Name: TaskMaster
Returns status of the event queue, updates window events
Push: Result Space (W); Event Mask (W); Event Record (L)
Pull: Extended Event Code (W)
Errors: $0E03

Memory Manager Calls

Function: $0902
Name: NewHandle
Makes a block of memory available to your program
Push: Result Space (L); Block Size (L); UserID (W); Attributes (W); Address of Block (L)
Pull: Block's Handle (L)
Errors: $0201, $0204, $0207

Function: $2002
Name: HLock
Locks and sets a specific handle to a purge level of 0
Push: Handle (L)
Pull: Nothing
Errors: $0206

Function: $2802
Name: PtrToHand
Copies a number of bytes from a specific memory address to a handle
Push: Source Address (L); Destination Handle (L); Length (L)
Pull: Nothing
Errors: $0202, $0206

Chapter 11

Controls

Controls are things you can put into dialog boxes or windows to perform specific functions. In addition, they have their own identities and allow a user to interact with a program using standards that are maintained in all Apple applications.

The nicest part about controls, like just about everything

else in the Toolbox, is that most of the work relating to them is done for you. You simply define the control, stick it in a window, and your work is done. When you consider that description, a chapter on controls might seem to be useless. Yet, there's a lot of information about controls that doesn't exactly fit under any other rubric. Hence, this chapter is full of information about controls.

This chapter doesn't focus on the Control Manager, but instead concerns itself with the individual controls themselves. The chapter on the Dialog Manager gives dialog boxes a thorough going-over. But much more can be said about controls inside the dialog box. Therefore, this chapter has two areas of concentration:

- The Control Manager
- Controls

The first part of this chapter provides some general information about the Control Manager (one of the more important tool sets). Then the chapter turns to techniques for customizing the standard controls already defined in the Toolbox so that they are best suited to your programs. At the end of this chapter you will find examples of the Control Manager being used to set or change the value of a control.

The Control Manager

The Control Manager is one of the more important, as well as obscure, tool sets. The following two tool sets rely upon the Control Manager in order to operate properly:

- Window Manager
- Dialog Manager

The reason for this is that both of these tool sets use controls. All the items inside a window—the grow and zoom boxes and the scroll bars—as well as the items in a dialog box are controls. The Control Manager is the tool set whose job it is to manipulate those controls. You can choose from a list of predefined controls: buttons, radio buttons, check boxes, LineEdit boxes, and so on. Or, by using the Control Manager, you can create custom controls to use in your programs.

Many of the functions of the Control Manager are called internally by other tool sets. For example, the Window Manager must access certain Control Manager functions to place the proper controls into a window. And when you set up a dialog box, it's the Control Manager that handles the intricacies of defining the controls and maintaining their values. As will be seen in a later section, many of the Dialog Manager's functions have similar, corresponding Control Manager functions, some of which are called internally by the Dialog Manager.

Before you start the Control Manager, the following tool sets should already have been started:

- Tool Locator
- Memory Manager
- Miscellaneous tool set
- QuickDraw II
- Event Manager
- Window Manager

To start the Control Manager the CtlStartUp call is made. You'll need to send the Toolbox your program's User ID, and set aside one page ($100 bytes) of direct page space.

In machine language:

```
pushword     UserID    ;push our user id
pushword     DPage     ;push direct page location
_CtlStartUp
jsr          ErrChk    ;check for errors
```

In C:

```
CtlStartUp(UserID, GetDP(0x100)); ErrChk();
```

In Pascal:

```
CtlStartUp(UserID, GetDP($100)); ErrChk;
```

The GetDP call in the C and Pascal examples is described in the MODEL program, illustrated in Chapter 6.

The only error being checked for after the CtlStartUp call is $1001, meaning the Window Manager has not been initialized. So when you're writing applications, it's a good idea to start up the Window Manager before the Control Manager. Also, as is true with all other tool sets, the Control Manager functions better if its allocated direct page space is page-aligned. (See the information on the NewHandle function in Chapter 7 for more information.)

To shut down the Control Manager, a call is made to CtlShutDown.

In machine language:

```
_CtlShutDown
```

In C:

```
CtlShutDown();
```

In Pascal:

```
CtlShutDown;
```

Be careful to shut down the Window Manager before making the above calls. If you're simply shutting down all the tool sets to quit a program, then the order isn't that crucial. Still, it's a good idea to shut down the Window Manager first. You may wonder why this practice is recommended. The reason is that the Window Manager is responsible for disposing of windows (and dialog boxes) containing controls. Therefore it's a good idea to shut it down first. This assures that there are no controls left on the screen when CtlShutDown is called. (CtlShutDown does not remove the controls, so when the Window Manager makes the call to the Control Manager to remove the controls, an error results.)

Shut down tool sets following the reverse of the order in which they were started up.

Controls

The Control Manager maintains several built-in controls. All the items in a window that manipulate the window are controls. Others managed by the Control Manager include the following items, which you can specify in a dialog box:

- Buttons
- Check boxes
- Radio buttons
- Scroll bars
- Edit lines
- Grow box

For each type of control there is a control record. This record contains information about the control:

- The window to which it belongs
- Pointers to its action procedure
- Pointer to a color table

It also contains information defined by your program when the control was initially put on the screen, or as maintained by the Control Manager as you are manipulating the control.

The following sections detail each type of control. This information is provided to enhance information already presented in Chapter 10. For example, the following sections contain information about certain controls' ItemValue and ItemFlag, and how these values can be manipulated to give your programs their own unique look. Plus, there's information about changing the default color of a control.

The following built-in controls can be specified as part of a dialog box via the NewDItem or GetNewDItem calls of the Dialog Manager. NewDItem specifies each aspect of the control one at a time, whereas GetNewDItem uses a template of values.

In summary, GetNewDItem sets up a call to NewDItem. NewDItem, on the other hand, contacts the Control Manager to set up the control. The Control Manager manipulates the information further and calls NewControl, which actually sets up the control record and assigns the control to a particular window. NewControl may do further initializing depending upon the type of control.

Push button. Push buttons always perform some action, or they can activate something. Unlike other controls that can be switched on or off or positioned in some manner, when a push button is clicked by the mouse, it immediately causes something to happen (usually it closes a dialog box).

Table 11-1 shows the items specified when a push button is defined. These items would either be individually specified via the NewDItem function, or using a template with the GetNewDItem function.

Table 11-1. Items Specified When Push Button Is Defined

Name	Size	ButtonItem Value
ItemID	Word	The button's ID
ItemRect	Word	Upper left Y position of the button (MinY)
	Word	Upper left X position of the button (MinX)
	Word	Usually 0
	Word	Usually 0
ItemType	Word	$000A (10 decimal)
ItemDescr	Long	Pointer to string inside the button
ItemValue	Word	Always 0
ItemFlag	Word	Determines visibility and type of button
ItemColor	Pointer	A table defining the button's color

ItemID. ItemID assigns a unique value to the button. A value of $0001 defines the button as the default button of the dialog box. The default button has a double outline. Pressing Return is the same as clicking the default button.

An ItemID of $0002 defines the default Cancel button, which is equivalent to pressing the Escape key. Other values can be used simply to define a typical push button.

ItemRect. The ItemRect of the button defines its location and size relative to the upper left corner of the dialog box (position 0,0). Normally, only the first two words of this rectangle are specified; the last two can be zeros. The Control Manager will fill in the other corner based on the size of the text inside the button.

Later in this chapter, an example of a button is shown with all four values defined. Even though the second two words need not be actual values, the Control Manager will still create a push button (though of a nonstandard size), and will still center the text within that button.

within that button.

ItemType. The ItemType for a button is $000A, or 10 decimal.

Instead of using a raw number, check your language's support files for predefined symbol names that can greatly improve the readability of your program. For example, when you include the <dialog.h> header file in your C programs, you can use the defined constant called buttonItem rather than the number 0x000a (hex) or 10 (decimal).

ItemDescr. ItemDescr is a long-word pointer to the string to be placed inside the button. The string should be rather short, as anything longer than one or two words is considered an essay. When that's the case you should consider whether the button is appropriate. The button's string should start with a count byte (a Pascal string).

ItemValue. ItemValue should always be a word of 0. A button does not require an item value.

ItemFlag. ItemFlag is a word describing whether the button will be visible or invisible, and it also determines what type of frame the button will have. Only the LSB (lower byte) of this word holds any value; the upper byte should always be 0.

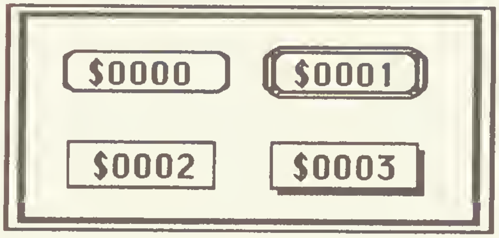
Bit 7 of the ItemFlag word determines the visibility of the button. When bit 7 is set to 1 (a value of $0080), the button is invisible. When bit 7 is reset to 0, the button is visible. There are Dialog Manager and Control Manager functions that will change a button's visibility after it has been created. (Note that there is a difference between a visible button and one that is disabled. See below.)

Bits 0 and 1 of ItemFlag determine the style of the button's frame, or outline. Buttons can have square or round corners, and they can have a double outline or a drop shadow, all depending on how these bits are set.

Table 11-2. Style of Button's Frame

Bit			
1	**0**	**Hex Value**	**Meaning**
0	0	$0000	Typical round-cornered button
0	1	$0001	Round-cornered button with double border
1	0	$0002	Square button
1	1	$0003	Square button with a drop shadow

Figure 11-1. The Four Types of Buttons



The default button for a dialog box uses a bit pattern of $0001. Other bit patterns for ItemFlag can be used to create different-shaped buttons. However, Apple advises against using the double-border pattern ($0001) on buttons other than the default button.

ItemColor. ItemColor is a long-word pointer to a color table for the button. The color table can be used to specify colors other than black and white for the button's parts. For example, the button's text could be green on pink and the button could be gray on blue.

Table 11-3 describes the color table used for a push button (and pointed to by ItemColor).

Table 11-3. Push Button Color Table

Offset	**Size**	**Parameter**	**Bits**		
			15–8	**7–4**	**3–0**
$00	Word	SimpOutline	0	OUT	0
$02	Word	SimpNorBack	0	BG	0
$04	Word	SimpSelBack	0	BG	0
$06	Word	SimpNorText	0	BG	FG
$08	Word	SimpSelText	0	BG	FG

OUT = Outline color
BG = Background color
FG = Foreground color
0 = Always zero

The individual bit positions in each word of the color table are used to specify which colors are used to color each part of the button. In the 320 mode, all four bit positions (7–4 or 3–0) are used to specify one of 16 different colors. In the 640 mode, only bits 4 and 5, or bits 0 and 1, are used to specify color. Be careful to note which values of the word (bitwise) are used and which aren't.

SimpOutline. SimpOutline describes the color of the button's outline.

SimpNorBack. SimpNorBack is the background color of the button when the button is not being pressed.

SimpSelBack. SimpSelBack is the background color of the button when the button is being pressed.

SimpNorText. SimpNorText is the color of any text inside the button when the button is not being pressed. The background color of the text is specified in bits 7–4 and the foreground color in bits 3–0.

SimpSelText. SimpSelText is the color of any text inside the button when the button is being pressed. The background color of the text is specified in bits 7–4 and the foreground color in bits 3–0.

The following creates a rather interesting colored button (in 320 mode). You might want to include a color table such as this with a program that uses the colorful menu bar example from Chapter 8.

```
ButtonColorT  dc   i2'%0000000000110000'
              dc   i2'%0000000001010000'
              dc   i2'%0000000011010000'
              dc   i2'%0000000001110110'
              dc   i2'%0000000010001001'
```

Notice how similar this is to setting the color table for a window as described in Chapter 9.

Check box. A check box represents a condition, either on or off. Clicking in a check box doesn't automatically turn it on, or activate it. Instead, its ItemValue must be changed either through the SetDItemValue call in the Dialog Manager, or via Control Manager calls as outlined in a later section of this chapter. (This was covered briefly in the previous chapter.) When you click the mouse in a check box, it should become checked if it wasn't already, or it should become unchecked if it was. This logic is supplied by your program.

Check boxes have a line of text beside them. Unlike static text items, the text by a check box is defined along with other attributes of the check box. Therefore, the position of the check box on the screen should account for any text just to the right of it.

Table 11-4 shows the values used to define a check box:

Table 11-4. Values Used to Define a Check Box

Name	Size	CheckItem Value
ItemID	Word	The check box's ID
ItemRect	Word	Upper left Y position of the check box (MinY)
	Word	Upper left X position of the check box (MinX)
	Word	Zero
	Word	Zero
ItemType	Word	$000B (11 decimal)
ItemDescr	Long	Pointer to check box's title string
ItemValue	Word	$0000 for open, any other value for selected
ItemFlag	Word	Determines visibility
ItemColor	Pointer	A table defining the box's color

ItemID. The ItemID of a check box can be any value used to identify the checkbox uniquely. You could specify an ItemID of $0001 or $0002; it isn't recommended, however. This would clash with the rules set down in Apple's Human Interface Guidelines. Only a push button should be the default button in a dialog box, so only a push button should have an ItemID of $0001 or $0002.

ItemRect. ItemRect, like a button, defines the location of the check box relative to the upper left corner of the dialog box. Any text appearing next to the check box will be to the right of the check box. As with a button, keep the text brief.

ItemType. The ItemType of a check box is $000B, or 11 decimal.

ItemDescr. ItemDescr is a long-word pointer to the string appearing next to the check box. The string should start with a count byte.

ItemValue. ItemValue indicates the initial value of the check box. If ItemValue is 0, the check box is empty, or unchecked. If ItemValue is any nonzero value, the check box is checked, indicating that whatever state the check box is monitoring is presently selected, or active.

ItemFlag. A check box's ItemFlag holds the same meaning that it does for a push button: It determines whether the check box will be visible or invisible. A value of $0080 means the check box will be invisible, while a value of $0000 means the check box will be visible.

ItemColor. ItemColor is a long-word pointer to a color table for the check box. Table 11-5 describes the items in a check box's color table.

Table 11-5. Items in Check Box's Color Table

Offset	Size	Parameter	Bits		
			15–8	7–4	3–0
$00	Word	CheckReserved	0	0	0
$02	Word	CheckNorColor	0	BG	FG
$04	Word	CheckSelColor	0	BG	FG
$06	Word	CheckTitleColor	0	BG	FG

BG = Background color
FG = Foreground color
0 = Always zero

The same information for a push button's color table (regarding bit positions) holds true for this and all succeeding color tables. Remember that the 320 mode is much more colorful than the 640 mode.

CheckReserved. CheckReserved should be a word of 0. Presumably Apple has something clever in mind for this value and just won't let us know what it means.

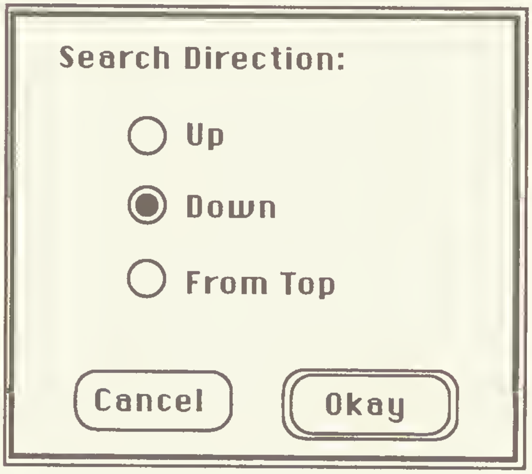
CheckNorColor. CheckNorColor is the color of the check box when it's not highlighted or selected.

CheckSelColor. CheckSelColor is the color of the check box when it's highlighted or selected. An example of color usage would be to specify bits 7–4 to show a different color (say, red) for a selected check box.

CheckTitleColor. CheckTitleColor is the background and foreground color of the check box's title string at all times. (The title does not change as the box changes.)

Radio button. Radio buttons are among the most useful types of controls. Yet they are also easily misunderstood. With radio buttons, only one in a series can be selected at a time—and one of the series must be on. Figure 11-2 gives an example of a good use for radio buttons.

Figure 11-2. Row of Three Radio Buttons: Up, Down, and From Top



Search Direction:

○ Up

● Down

○ From Top

Why call them radio buttons? The analogy Apple gives is that of an old car radio. The buttons on the radio were used to switch from one preselected radio station to another. Only one of the buttons could be down at a time—you couldn't listen to more than one station. When you pushed one in, any other button that was pressed in would be automatically released.

Radio buttons should be used in an application when one of several options must be selected, but not more than one. If it's possible to choose more than one option, check boxes should be used.

You can specify which radio button is to be on when the dialog box is created. However, as with other items in a dialog box, further manipulation of the radio buttons is up to your program. (Refer to the COLOR program from Chapter 10 for a good example of radio button manipulation.)

Table 11-6 shows the values used to define a radio button.

Table 11-6. Values Used to Define Radio Buttons

Name	**Size**	**RadioItem Value**
ItemID	Word	The radio button's ID
ItemRect	Word	Upper left Y position of the button (MinY)
	Word	Upper left X position of the button (MinX)
	Word	Zero
	Word	Zero
ItemType	Word	$000C (12 decimal)
ItemDescr	Long	Pointer to radio button's title string
ItemValue	Word	$0000 for open, any other value for selected
ItemFlag	Word	Determines visibility and family number
ItemColor	Pointer	A table defining the button's color

ItemID. The ItemID of a radio button, as with a check box, can be any value except $0001 or $0002. A *family number* can be given to a radio button via its ItemFlag value. This family number is used to group radio buttons according to their function, and to ensure that only one radio button within a particular family is on at a time. (The Control Manager will actually prevent you from activating more than one radio button at a time. See the ItemFlag description below.)

ItemRect. ItemRect defines the radio button's location relative to the upper left corner of the dialog box. Any text appearing next to the radio button will be to its right.

ItemType. The radio button ItemType is $000C, or 12 decimal.

ItemDescr. ItemDescr is a long-word pointer to a Pascal string to appear next to the radio button.

ItemValue. ItemValue indicates the initial value of the radio button. As with a check box, when ItemValue is 0, the radio button is unselected, and when ItemValue is any nonzero value, the radio button is highlighted.

ItemFlag. ItemFlag determines the visibility of the radio button as well as its family number. Bit 7 of the ItemFlag word determines visibility. When this bit is set to 1, the radio button is invisible; when bit 7 is reset to 0, the radio button is visible. The remainder of the bits in this word (bits 6–0) specify the family number of the button. Values in the range $0000–$007F can be used for up to 128 family numbers.

ItemColor. ItemColor is a long word pointer to a color table for the radio button.

Table 11-7. Meaning of Bits Within ItemColor

Offset	Size	Parameter	Bits		
			15–8	7–4	3–0
$00	Word	RadioReserved	0	0	0
$02	Word	RadioNorColor	0	BG	FG
$04	Word	RadioSelColor	0	BG	FG
$06	Word	RadioTitleColor	0	BG	FG

BG = Background color
FG = Foreground color
0 = Always zero

RadioReserved. RadioReserved is a word of 0, reserved for some future date. Perhaps Apple will design a three-dimensional radio button selected with this value.

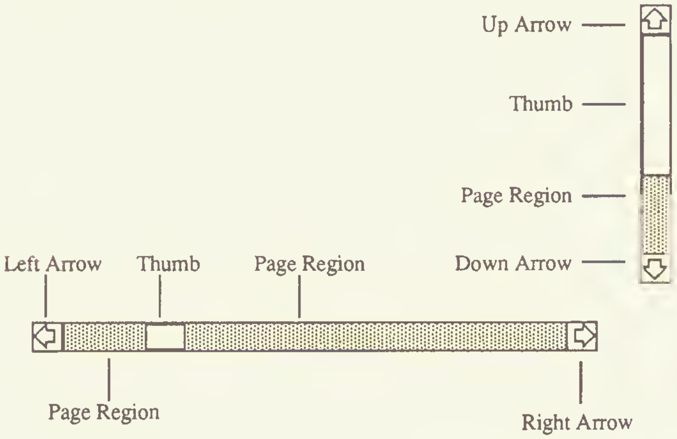
RadioNorColor. RadioNorColor is the color of the radio button when it's not highlighted or selected.

RadioSelColor. RadioSelColor is the color of the radio button when it is highlighted or selected.

RadioTitleColor. RadioTitleColor is the background and foreground color of the radio button's title string.

Scroll bar. You may not think of scroll bars as controls, but they are. They're just like buttons, check boxes, and radio buttons. They're usually used with windows. However, they can be used for other purposes if you know how to manipulate them.

Figure 11-3. Diagram of Scroll Bar with Associated Terms



The scroll bar is the most complex type of control you can define. The Window Manager uses scroll bars in windows to scroll an area of data. However, if you want to put a scroll bar into a dialog box just to see what it's like, you'll need to know the information provided by Table 11-8.

Table 11-8. Information Required to Define a Scroll Bar

Name	Size	ScrollBarItem Value
ItemID	Word	The scroll bar's ID
ItemRect	Word	Upper left Y position of the scroll bar (MinY)
	Word	Upper left X position of the scroll bar (MinX)
	Word	Lower right Y position of the scroll bar (MaxY)
	Word	Lower right X position of the scroll bar (MaxX)
ItemType	Word	$000D (13 decimal)
ItemDescr	Long	Zero, or a pointer to an action procedure
ItemValue	Word	Data size minus view size (greater than 0)
ItemFlag	Word	Determines visibility and scroll bar items
ItemColor	Pointer	A table defining the scroll bar's color

ItemID. ItemID is a value used to identify the scroll bar.

ItemRect. ItemRect defines the scroll bar's location in the dialog box (or window), relative to the dialog box's upper left corner (local coordinates). The two words indicating the lower right corner of the scroll bar take on significance here and must be specified. Together the four word values create the rectangle into which the Control Manager will squeeze the scroll bar.

By adjusting the corner positions of the scroll bar, you can have a very skinny scroll bar, or one that's terribly fat. Because a scroll bar is a predefined control, you can subtly change the way it looks to use it as a custom control in your programs.

ItemType. The ItemType of a scroll bar is $000D, or 13 decimal.

ItemDescr. ItemDescr is the long-word address of a scroll bar action procedure used to control the scroll bar. A long word of 0 can be used to specify the default procedure.

ItemValue. ItemValue indicates the position of the *thumb* in the scroll bar. The higher the value, the further along in position the thumb will be (with the origin at the top or far left of the scroll bar, depending upon the scroll bar's orientation).

ItemFlag. ItemFlag determines the visibility of the scroll bar, as well as the orientation of the scroll bar and what types of arrows it will have. (The thumb and page regions of the scroll bar are included standard, but the up/down or right/left arrows are considered optional.) As with other ItemFlag values, only bits 7 through 0 hold any significant value in this word. All other bits should be reset to 0.

Table 11-9 shows the meanings of the bit positions in a scroll bar's ItemFlag.

Table 11-9. Meaning of Bit Positions in Scroll Bar's ItemFlag

Bit	Meaning if Set
7	Scroll bar is invisible
6	Nothing (should always be 0)
5	Nothing (should always be 0)
4	Scroll bar is horizontal (right to left)
3	Scroll bar will have a right arrow
2	Scroll bar will have a left arrow
1	Scroll bar will have a down arrow
0	Scroll bar will have an up arrow

If bit 4 above is reset to 0, the scroll bar will be vertical, or up and down.

You can specify arrows either in one or both directions (up/down, left/right) for your scroll bar. It's possible to specify a left/right arrow with an up/down scroll bar, even though it's wrong. Your program will not crash, but the scroll bar will be updated improperly and your dialog box will fill with random graphics. In other words, it's ill-advised.

So, to specify a full-on vertical scroll bar with both arrows, an ItemFlag of $0003 is used. For a full-on horizontal scroll bar, an ItemFlag of $001C can be used.

ItemColor. ItemColor is a long word pointer to the scroll bar's color table as shown below.

Table 11-10. Meaning of Bits Within ItemColor

Offset	Size	Parameter	Bits		
			15–8	7–4	3–0
$00	Word	ScrollOutline	0	OUT	0
$02	Word	ArrowNorColor	0	BG	FG
$04	Word	ArrowSelColor	0	BG	FG
$06	Word	ArrowBackColor	0	BG	0
$08	Word	ThumbNorColor	0	BG	0
$0A	Word	ScrollReserved	0	0	0
$0C	Word	PageRgnColor	PAT	COL1	COL2
$0E	Word	InactiveColor	0	BG	0

OUT = Outline color
BG = Background color
FG = Foreground color
PAT = Color pattern
0 = Always zero

ScrollOutline. ScrollOutline is the outline color of the scroll bar, arrow boxes, and thumb.

ArrowNorColor. ArrowNorColor is the color of the arrow outline and background when an arrow is not being selected by the mouse.

ArrowSelColor. ArrowSelColor is the color of the arrow (filled) and background when the arrow is selected by the mouse. A good method of setting this and the previous color value is to reverse them: Use the foreground color for ArrowNorColor and the background color for ArrowSelColor, and vice versa.

ArrowBackColor. ArrowBackColor is the interior color of the arrow when it is not selected.

ThumbNorColor. ThumbNorColor is the color of the thumb's interior.

ScrollReserved. ScrollReserved is a word of 0, reserved for some secret future use.

PageRgnColor. PageRgnColor is the color of the page region in the scroll bar. The MSB of this word determines whether a dithered pattern is to be used. The LSB of the word contains either the solid color with which to fill the page region, or two colors to use for dithering.

If bit 8 is set, dithering takes place. The page region is filled with a checked pattern of both the colors specified in bits 7–4 and 3–0.

If bit 8 is reset to 0, the page region is filled with the solid color pattern indicated by the color specified in bits 7–4. Bits 3–0 should all be reset to 0.

Bits 15–9 of the PageRgnColor value should always be 0.

InactiveColor. InactiveColor is the color of the scroll bar when it has been deactivated (dimmed).

Edit lines. Edit lines are controls that allow a user to type a line of text into a dialog box. Edit lines are best used when the information needed by your program cannot be obtained by using a button or list of items.

Any text typed at the keyboard will appear in the edit box. Additionally, because of the LineEdit tool set, the text inside the edit line can be edited, selected with the mouse, cut, pasted, deleted, or copied to a special edit line clipboard (maintained by the Toolbox) using the standard editing keys. (See Appendix A for more on editing.)

Any key pressed will appear in the edit line. When Return is pressed, the default button of the dialog takes over and the dialog box vanishes. Because of this, if more than one edit line appears in a dialog box, the Tab key is pressed to switch between one edit line item and another. If a number of edit lines are in a single dialog box, the Tab key can be pressed repeatedly until the insert cursor is in the desired edit line.

If a default button is not defined, the Return character (an inverse question mark in the system font, or simply a blank) is displayed in the edit line just like any other character.

The first edit line defined, either by the NewDItem or GetNewDItem functions or first in a template of items for the GetNewModalDialog call, is the first edit line created and placed into the dialog. The cursor appears in the first defined edit line box. The ItemID of the edit line has nothing to do with its order.

The items listed in Table 11-1 are used to define an edit line.

Table 11-11. Information Required to Define an Edit Line

Name	Size	EditLine Value
ItemID	Word	The EditLine's ID
ItemRect	Word	Upper left Y value of EditLine's box (MinY)
	Word	Upper left X value of EditLine's box (MinX)
	Word	Lower right Y value of EditLine's box (MaxY)
	Word	Lower right X value of EditLine's box (MaxX)
ItemType	Word	$0011 (17 decimal)
ItemDescr	Long	Pointer to string inside the EditLine, or buffer
ItemValue	Word	Max characters to be typed (up to 255)
ItemFlag	Word	Determines visibility
ItemColor	Pointer	Always 0

ItemID. The ItemID is a unique number used to identify the edit line. Its value is really unimportant because editing and entering text takes place automatically.

ItemRect. ItemRect defines a rectangle indicating the size and position of the edit line's input box in local coordinates. The length of the box (left to right) depends on the number of characters the user should be allowed to enter (and, indirectly, depends on the system font as well). The height of the box must be at least 15 pixels—anything less and text inside the edit line will not be visible.

The height of the edit line's box really depends on the size of the font used by the Dialog Box. For a smaller font, logically, a box of less than 15 pixels in height could be used. Likewise, if an exceptionally large font were being used, a height taller than 15 pixels would be required.

ItemType. The ItemType for an edit line is $0011, or 17 decimal.

ItemDescr. ItemDescr points to either a string of text that may be edited, or an empty buffer into which typed text will be placed. ItemDescr must point to something, either an empty buffer or a string of text. If ItemDescr is the address of a Pascal string of text, that text appears as selected when the Control Manager draws the edit line.

ItemValue. ItemValue determines how many characters are allowed inside the edit line. Only the number of characters specified by ItemValue can be typed into the edit line, and no more. ItemValue also indirectly indicates the size of the string pointed to by ItemDescr.

ItemFlag. ItemFlag can be one of two values. When Itemflag is $0080, the edit line's box is invisible, but the text can still be seen. When ItemFlag is 0, EditLine's box is drawn.

The edit line control does not use a color table, so its value should be reset to a long word of 0.

Changing Colors

Almost every control can take advantage of color. Your dialog boxes can be made colorful simply by specifying a color table pointer and filling the table with the desired values for each control. But some confusion can arise in referring to color tables as used by controls and color tables used by QuickDraw.

It should be pointed out that the color tables used when defining a control are the same as the color tables used by QuickDraw.

QuickDraw defines a color table from which certain colors are selected. For example, in the 320 mode, QuickDraw sets up a color table with 16 separate colors. Each color is defined according to the intensity of its red, green, and blue attributes. So, in a QuickDraw color table, color number 5 in that table may be set to dark green.

In the color tables used by controls, the values referred to are the values in the QuickDraw color tables. So if the current color table as used by QuickDraw has 16 values and number 5 is dark green, then when you specify a value of 5 in your control table, it takes on the color dark green. In fact, all the pixels on the super-high-resolution graphics display on the Apple IIGS work this way: They aren't fixed color values; they're simply index numbers into a color table.

Table 11-12 shows how QuickDraw assigns color values in the standard 320-mode color table. The control value and color indicate the value specified in a control's color table and the color that value represents. Use this table to determine which values in your control's color tables will take on which colors (using the standard color table in the 320 mode).

Table 11-12. Color Values

QuickDraw Number	Color	Control Value	
		Binary	Hexadecimal
0	Black	0000	$0
1	Dark gray	0001	$1
2	Brown	0010	$2
3	Purple	0011	$3
4	Blue	0100	$4
5	Dark green	0101	$5
6	Orange	0110	$6
7	Red	0111	$7
8	Beige	1000	$8
9	Yellow	1001	$9
10	Green	1010	$A
11	Light blue	1011	$B
12	Lilac	1100	$C
13	Periwinkle	1101	$D
14	Light gray	1110	$E
15	White	1111	$F

A control's color table can be changed or altered to suit your personal tastes and whatever is in vogue.

Panic Button

The following code (Programs 11-1 to 11-3) shows how a push button's size and color can be manipulated to create a very large panic button. These examples are not complete programs. The code represents a panic button subroutine (to be called at the appropriate time) that you can place into your own programs.

Program 11-1. Panic Button in Machine Language

```
*-------------------------------*
* PANIC Button Dialog Box       *
*-------------------------------*

:Equates...

DialogHeight    equ     100
DialogWidth     equ     110
ItemDisable     equ     $8000
StatText        equ     $0F
ButtonItem      equ     $0A

:Start of Routine...
```

```
Panic   pea     $0000           ;long word result space
        pea     $0000
        pushlong #DialogRecord
        _GetNewModalDialog
        jsr     ErrChk

        pulllong DialogPtr      ;get dialog pointer

;Now wait until the button is clicked

Wait    pea     $0000           ;result space
        pea     $0000           ;filter routine (long pointer)
        pea     $0000
        _ModalDialog            ;get dialog events

        pla                     ;get results
        cmp     #$1             ;was it the panic button?
        bne     Wait            ;keep waiting if not

        pushlong DialogPtr      ;we're done, close the dialog
        _CloseDialog

        rts                     ;return, done

;----Data Storage----

DialogPtr       ds 4

DialogRecord    anop
        dc      i2'(190-DialogHeight)/2'
        dc      i2'(320-DialogWidth)/2'
        dc      i2'(190-DialogHeight)/2+DialogHeight'
        dc      i2'(320-DialogWidth)/2+DialogWidth'
        dc      i2'TRUE'
        dc      i4'0'
        dc      i4'TextRecord'
        dc      i4'ButtonRecord'
        dc      i4'0'
TextRecord      anop
        dc      i2'2'
        dc      i'5,5,15,105'
        dc      i2'ItemDisable+StatText'
        dc      i4'TextString'
        dc      i2'0'
        dc      i2'0'
        dc      i4'0'

TextString      anop
        dc      i1'15'
        dc      c"It's time to..."

ButtonRecord    anop
        dc      i2'1'
        dc      i'25,5,95,105'
        dc      i2'ButtonItem'
        dc      i4'ButtonString'
        dc      i2'0'
        dc      i2'0'
        dc      i4'ColorTable'
```

```
ButtonString    anop
        str     'Panic'

ColorTable      anop
        dc      i2'%0000000001010000'
        dc      i2'%0000000011110000'
        dc      i2'%0000000001110000'
        dc      i2'%0000000011110000'
        dc      i2'%0000000001110000'
```

Program 11-2. Panic Button in C

```
/*------------------------------*
 * PANIC Button Dialog Box      *
 *------------------------------*/

#define DialogHeight    100
#define DialogWidth     110

ItemTemplate    TextRecord = {
                    2,
                    5, 5, 15, 105,
                    itemDisable+statText,
                    "\pIt's time to...",
                    0, 0. NULL
};

BttnColors  ColorTable = {
                    0x0050,
                    0x00f0,
                    0x0070,
                    0x00f0,
                    0x0070
};

ItemTemplate    ButtonRecord = {
                    1,
                    25, 5, 95, 105,
                    buttonItem,
                    '\pPanic',
                    0, 0, &ColorTable
};

DialogTemplate  DialogRecord = {
                    (190 - DialogHeight) / 2,
                    (320 - DialogWidth) / 2,
                    (190 - DialogHeight) / 2 + DialogHeight,
                    (320 - DialogWidth) / 2 + DialogWidth,
                    TRUE,
                    NULL,
                    &TextRecord.
                    &ButtonRecord,
                    NULL
};

Panic()
{
    GrafPortPtr DialogPtr;
    DialogPtr = GetNewModalDialog(&DialogRecord);   ErrChk();
    while (ModalDialog(NULL) != 1);     /* Wait for PANIC button */
    CloseDialog(DialogPtr);             /* Then close the dialog */
}
```

Program 11-3. Panic Button in Pascal

```
{ *-----------------------------*
  * PANIC Button Dialog Box     *
  *-----------------------------* }

PROCEDURE Panic;

CONST   DialogHeight    = 100;
        DialogWidth     = 110;

VAR     TextRecord:     ItemTemplate;
        ButtonRecord:   ItemTemplate;
        ButtonColors:   ControlColorTbl;
        DialogRecord:   DialogTemplate;
        DialogPort:     DialogPtr;
        TextString:     String;
        ButtonString:   String;

BEGIN

    TextString := 'It's time to...';
    ButtonString := 'Panic';

    WITH TextRecord DO BEGIN
        ItemID      := 2;
        SetRect     (ItemRect, 5, 5, 15, 105);
        ItemType    := ItemDisable+StatTextItem;
        ItemDescr   := @TextString;
        ItemValue   := 0;
        ItemFlag    := 0;
        ItemColor   := nil;
    END;

    WITH ButtonColors DO BEGIN
        SimpOutline := $0050;
        SimpNorBack := $00f0;
        SimpSelBack := $0070;
        SimpNorText := $00f0;
        SimpSelText := $0070;
    END;

    WITH ButtonRecord DO BEGIN
        ItemID      := 1;
        SetRect     (ItemRect, 5, 25, 105, 95);
        ItemType    := ButtonItem;
        ItemDescr   := @ButtonString;
        ItemValue   := 0;
        ItemFlag    := 0;
        ItemColor   := @ButtonColors;
    END;

    WITH DialogRecord DO BEGIN
        SetRect(boundsRect,
          (320 - DialogWidth) / 2,
          (190 - DialogHeight) / 2,
          (320 - DialogWidth) / 2 + DialogWidth,
          (190 - DialogHeight) / 2 + DialogHeight);
      dtVisible   := TRUE;
      dtRefCon    := 0;
      Item1Ptr    := @TextRecord;
      Item2Ptr    := @ButtonRecord;
      Terminator  := nil;
   END;

   DialogPort := GetNewModalDialog(DialogRecord);  ErrChk;
   REPEAT UNTIL ModalDialog(nil) = 1;           { Wait for PANIC button }
   CloseDialog(DialogPort);                     { Then close the dialog }
}
```

Changing Values

This section describes how a control can be manipulated after it has been defined. Some of the functions to manipulate a control are listed under the Dialog Manager; the ones listed below are under the Control Manager.

The Control Manager must have a handle to a control before that control can be manipulated (unlike the Dialog Manager, which requires only an ItemID). To get a control's handle, a call is made to the Dialog Manager's GetControlDItem function. Once the handle is obtained, the various Control Manager routines that manipulate a control can be used.

Controls can be highlighted or inactive (dimmed), visible or invisible, and selected or unselected. Make sure you know and understand these differences.

When a control is dimmed, it appears fuzzy in the dialog box. Clicking the mouse on the control will not activate it, just as selecting a dimmed menu item won't work.

A visible control is one you can see. A control can be made invisible, for example, when an option is not available, or as was demonstrated in Chapter 10, to page text.

Another attribute of a control is to be selected or unselected. This normally affects only two controls: the check box and radio button. When either of those buttons is selected, its button or box is filled, meaning whatever function it represents is active. (See the COLOR example from Chapter 10 for a demonstration.)

The following sections illustrate how the Control Manager can be used to dim, hide, or activate a control.

Dimming controls. The following routines will dim or highlight a control using the HiliteControl function in the Control Manager.

HiliteControl can specify whether a control is to be redrawn as normal or inactive, or whether a specific part code of the control can be individually highlighted. (The entire control is always redrawn each time HiliteControl is called.)

The parameter determining how the control is highlighted is referred to as HiliteState. It's a word-sized value, though only the least significant byte holds any meaning:

HiliteState Value	**Highlighting**
0	Control is highlighted
1–253	Only specified parts are highlighted
254	Reserved (not used)
255	Control is dimmed

Part codes are used to identify the individual parts of a control. In the normal operation of a DeskTop application, your program will probably never need to manipulate any individual part codes. (You'll either be dimming or highlighting the entire control.)

But, for the curious, Table 11-13 shows the part numbers defined for specific controls. Values 32–127 are available for your application's use. Any other value not listed is reserved.

Table 11-13. Controls' Part Numbers

Code		
Decimal	**Hexadecimal**	**Part**
0	$00	None
2	$02	Simple button
3	$03	Check box
4	$04	Radio button
5	$05	Up arrow
6	$06	Down arrow
7	$07	Page up
8	$08	Page down
9	$09	Static text
10	$0A	Grow box
11	$0B	Edit line
12	$0C	User item
13	$0D	Long static text
14	$0E	Icon
129	$81	Thumb

The following code can be used to dim a control.

In machine language:

```
pushlong          #0                ;long result space
pushlong          DialogPtr         ;dialog box port pointer
pushword          ItemID            ;the control's ItemID
_GetControlDItem                    ;Dialog Manager Call

pulllong          ControlHandle     ;return a handle to the control

pea               255               ;dim the control
pushlong          ControlHandle
_HiliteControl
```

In C and Pascal:

```
HiliteControl(255, GetControlDItem(DialogPtr, ItemID));
```

Conversely, the following code will highlight a dimmed control (or simply redraw a highlighted control).

In machine language:

```
pushlong          #0                ;long result space
pushlong          DialogPtr         ;dialog box port pointer
pushword          ItemID            ;the control's ItemID
_GetControlDItem

pulllong          ControlHandle

pea               0                 ;redraw the control normal
ushlong           ControlHandle
                  _HiliteControl
```

In C and Pascal:

```
HiliteControl(0, GetControlDItem(DialogPtr, ItemID));
```

Control visibility. The easiest way to make a control visible or invisible is by setting or resetting bit 7 of its ItemFlag. If bit 7 is reset to 0, the control is visible. If bit 7 is set to 1, the control is invisible.

The Dialog Manager functions HideDItem and ShowDItem can be used to alter the visibility of a control after it's been defined.

In machine language:

```
pushlong      DialogPtr     ;dialog box pointer
pushword      ItemID        ;ItemID of the control
_HideDItem                  ;render it invisible
jsr           ErrChk        ;test for error $150C (item not found)
```

In C and Pascal:

```
HideDItem(DialogPtr, ItemID);
```

To make a control visible, simply replace the above HideDItem functions with ShowDItem. Note that showing an item already visible, as well as hiding an item already hidden, has no effect.

To hide a control using the Control Manager, some extra steps are required. Actually, it's recommended you use the above Dialog Manager functions. However, if you're partial to the Control Manager, you'll need to call GetControlDItem (in the Dialog Manager) to return the control's handle, then perform either the Control Manager's HideControl or ShowControl function.

In machine language:

```
   pushlong          #0           ;long result space
   pushlong          DialogPtr    ;dialog box port pointer
   pushword          ItemID       ;the control's ItemID
   _GetControlDItem
;                                 ;keep the control handle on the stack
   _HideControl                   ;Hide it
```

In C and Pascal:

```
HideControl(GetControlDItem(DialogPtr, ItemID));
```

To show the control again, replace the HideControl functions above with ShowControl.

Chapter Summary

The following tool set functions were referenced in this chapter.

Function: $0210
Name: CtlStartUp
Starts the Control Manager
Push: UserID (W); Direct Page (W)
Pull: Nothing
Errors: $1001

Function: $0310
Name: CtlShutDown
Shuts down the Control Manager
Push: Nothing
Pull: Nothing
Errors: None

Function: $0910
Name: NewControl
Creates a control
Push: Result Space (L); Window Pointer (L); Control's Rectangle (L); Title String (L); Item Flag (W); Initial Value (W); Extra Parameter 1 (W); Extra Parameter 2 (W); Definition Procedure (L); RefCon (L); Color Table (L)
Pull: Control Handle (L)
Errors: None

Function: $0E10
Name: HideControl
Hides a control, making it invisible
Push: Control Handle (L)
Pull: Nothing
Errors: None

Function: $0F10
Name: ShowControl
Shows a control, making it visible
Push: Control Handle (L)
Pull: Nothing
Errors: None

Function: $1110
Name: HiliteControl
Highlights or dims all or part of a control
Push: HiliteState (W); Control Handle (L)
Pull: Nothing
Errors: None

Dialog Manager Calls

Function: $0D15
Name: NewDItem
Places a control into a dialog box
Push: Dialog Pointer (L); ItemID (W); Rectangle pointer (L); ItemType (W); Item Descriptor (L); ItemValue (W); Item Flag (W); Color Table Pointer (L)
Pull: Nothing
Errors: $150A, $150B

Function: $1E15
Name: GetControlDItem
Returns a control handle for a dialog box item
Push: Result Space (L); Dialog Pointer (L); ItemID (W)
Pull: Control Handle (L)
Errors: $150C

Function: $2215
Name: HideDItem
Hides a control in a dialog box, rendering it invisible
Push: Dialog Pointer (L); ItemID (W)
Pull: Nothing
Errors: $150C

Function: $2315
Name: ShowDItem
Makes an item or control in a dialog box visible
Push: Dialog Pointer (L); ItemID (W)
Pull: Nothing
Errors: $150C

Function: $2F15
Name: SetDItemValue
Changes the value of an item, or selects an item
Push: New Item Value (W); Dialog Pointer (L); ItemID (W)
Pull: Nothing
Errors: $150C

Function: $3215
Name: GetNewModalDialog
Creates a modal dialog using a template
Push: Result Space (L); Template (L)
Pull: Dialog Pointer (L)
Errors: Possible Memory Manager errors

Function: $3315
Name: GetNewDItem
Places an item or control into a dialog box using a template
Push: Dialog Pointer (L); Template (L)
Pull: Nothing
Errors: $150A, $150B

Memory Manager Calls

Function: $0902
Name: NewHandle
Makes a block of memory available to your program
Push: Result Space (L); Block Size (L); UserID (W); Attributes (W); Address of Block (L)
Pull: Block's Handle (L)
Errors: $0201, $0204, $0207

Chapter 12

Interrupts

Interrupts. The very word evokes trepidation in even the most experienced programmer. Now, before you flee to the next chapter in terror, you'll find that interrupts on the IIGS are not only an essential part of the computer, but they're also a lot of fun.

The first section of this

chapter cushions the introduction to interrupts for the programmer who hasn't experienced an ordeal with them yet. It also presents the various forms of interrupts and task-switching capabilities that come as standard equipment on the Apple IIGS.

A collection of sample programs are used as the basis of study throughout the chapter, and you ought to find them exceptionally interesting, or at the very least, entertaining.

> Since interrupts involve working at the hardware level of the computer, you have to work with them in machine language. This doesn't mean that you cannot work with interrupts from C or Pascal. You can. But in order to understand the workings of interrupts, a knowledge of machine language is required. If you're a C or Pascal fan, you can take the ideas and low-level routines from the example programs in this chapter and link them with your own programs.

This chapter will concentrate on exploring the Toolbox's role in working with interrupts.

What Are Interrupts?

An interrupt is a signal that causes the microprocessor to stop its work and momentarily switch to something else. That "something else" is called an *interrupt handler*, also known as an *interrupt service routine*. An interrupt handler takes only a split second of processor time to complete its work, and then the microprocessor returns to its previous task.

A familiar interrupt on the IIGS is the invocation of the control panel. Pressing Control–Open Apple–Escape freezes the current program and brings up a new one: the Classic Desk Accessory menu. When finished with the control panel, the program that was interrupted continues where it left off, as though nothing had ever happened. The keyboard is one part of the computer that can generate an interrupt.

In computers such as the Apple IIGS, in which many things seem to happen all at once, the ability to share slices of processor time among routines is what keeps things running smoothly. It also frees the programmer from having to watch for certain events at every turn of the program. Imagine what a pain in the flowchart it would be if you had to keep an eye on the mouse location, move the pointer around, update the screen underneath, and so on. Since the mouse can generate interrupts when it is moved, or when its button is pressed, mouse interrupt handlers take care of all the mouse-related business behind the scenes.

Another source of interrupts is the serial port. These interrupts come into play when you have a modem connected to the computer while data is racing through the phone line. Each time a character comes through the modem and into the computer's modem port, an interrupt signal is generated. This causes a serial port interrupt handler to investigate all the commotion. When the handler discovers a character waiting at the port, it snatches the character away into a buffer, where it will be processed when the modem program is ready for it. This ensures that no characters will be lost if the computer is busy working on some other task.

Interrupts play a very important role in the operation of the Apple IIGS, especially since they are far more significant to the workings of that computer than they have been to any of its predecessors. But the correct handling of interrupts is one of the most tenuous programming tasks the budding IIGS programmer will face. Fortunately, the Apple IIGS has a few Toolbox functions that make working with interrupts easier and safer.

> Safer? Well, let's just say that if your custom interrupt handler is incorrectly written, you might find that it does a great job of reformatting your hard disk, even if you weren't writing a disk utility.
>
> Careful, precise handling of interrupts is imperative. So pay strict attention to the rest of this chapter if you haven't been scared away yet.

Types of Apple IIGS Interrupts

In the previous section, three main sources of interrupts on the Apple IIGS were introduced: the keyboard, the mouse, and the serial port. These are considered external hardware interrupt sources since they're activated by influences outside of the computer.

The Apple IIGS has many internal interrupts as well, mostly related to circuitry in the machine. The following is a list of some of the interrupts that can occur in an Apple IIGS:

Type	**Example Interrupt Activity**
Reset	Turning on the computer
Reset	Control-Reset, Control–Open Apple–Reset, or Diagnostics
Abort	Memory fault error (from expansion RAM)
IRQ	Any keypress executed while the Event Manager is active
IRQ	Keyboard flush (Control–Open Apple–Delete)
IRQ	Desk Accessory menu (Control–Open Apple–Escape)
IRQ	Mouse movement or button press
IRQ	Serial port (register state changes, and so on)
IRQ	Firmware print spooling (buffer refresh)
IRQ	Video graphics controller (scan line, VBL, and so on)
IRQ	Ensoniq DOC (sound RAM refresh signal)
IRQ	Realtime clock (one second, quarter-second)
Software	BRK instruction encountered
Software	COP instruction encountered

Interrupts come in five basic flavors:

Interrupt	**Explanation**
IRQ	Maskable interrupt request
NMI	Nonmaskable interrupt
Software	Software interrupt (BRK or COP)
Reset	System reset interrupt
Abort	Memory access abort interrupt

Maskable interrupt request (IRQ). A maskable IRQ interrupt is generated by a peripheral card or some other type of hardware that is physically or logically connected to the computer. A mouse, keyboard, serial port, Ensoniq DOC, clock, video graphics controller (VGC), and other such interrupt source generates IRQ interrupts. These can be masked (ignored) by the processor if the interrupt disable bit in the processor's status register is set (with the SEI instruction). Using the CLI instruction clears the disable bit, which means the processor will resume handling interrupt requests.

Just for kicks, enter the following BASIC program into Applesoft BASIC and run it.

```
10 SEI = 120 : CLI = 88 : RTS = 96
20 POKE 768, SEI
30 POKE 769, RTS
40 CALL 768
```

Now, try to bring up the Classic Desk Accessory (CDA) menu by pressing Control–Open Apple–Escape. You'll find that it refuses to pop up. This is because the 65816 processor is set to mask the interrupts that the ADB (Apple DeskTop Bus) Keyboard Micro is sending whenever the CDA menu is requested.

Change the SEI in Line 20 to CLI and rerun the program.

As soon as you press the Return key after typing RUN, the CDA menu appears. This will be discussed in detail later in the chapter.

Nonmaskable interrupts. Although no built-in source exists, a nonmaskable interrupt (NMI) is supported by the Apple IIGS. A nonmaskable interrupt is just like an IRQ except that (as you might guess) the processor cannot mask it out. Some Apple II peripherals, such as a screen snapshot-to-printer card or a hardware diagnostic card, can generate NMIs.

Software interrupts. A software interrupt can be generated by executing a BRK or COP machine language instruction. In one sense, these are nonmaskable interrupts; even if the processor's interrupt disable flag is set (SEI), a BRK instruction is still performed. BRK is used mainly for debugging purposes to insert a programmable break point in your programs. COP is intended to kick a coprocessor card—a math coprocessor, for example—into action.

Reset interrupts. Reset interrupts are generated mainly by pressing Control-Reset, Control–Open Apple–Reset (reboot), or Control– Open Apple–Option–Reset (diagnostics), or by turning on the computer. A reset interrupt can be simulated through software by sending a command to the Apple DeskTop Bus, or by directly calling the reset handler code in ROM ($00FA62 in emulation mode).

Abort interrupts. The Apple IIGS currently does not make use of an abort interrupt even though it is supported. Aborts are generated when access is made to an off-limits portion of memory, something all multi-user computers employ to keep users from poking around in other people's memory space. Should the IIGS become a true multi-user computer, this police-style interrupt would be valuable for maintaining security.

When an Interrupt Occurs

Here's a brief rundown of what happens when the processor is interrupted (that is, as long as interrupts aren't being masked). Keep in mind that all of this happens within a few milliseconds:

- When the computer is interrupted, a program in the Apple IIGS ROM, the firmware interrupt manager, runs through a checklist of tasks to service the interrupt. It first determines which set of interrupt vectors should be used, depending on emulation mode. (These vectors are listed in Appendix B of *Mastering the Apple IIGS Toolbox,* available from COMPUTE! Books.)
- The processor speed kicks in to fast mode.
- The type of interrupt is then determined. If it's due to a BRK or COP instruction, one of the software interrupt handlers is called. If the handler is not installed, the user is sent directly to the Apple IIGS monitor.
- Machine-state information (that is, registers and flags) is saved at this point, before the serial port is tested to see whether it originated the interrupt. If it did, either *AppleTalk* or a serial port interrupt handler is called.
- Finally, if the interrupt wasn't due to a software instruction or activity at the serial port, the rest of the machine-state information is saved, and then all the other internal interrupt sources (the clock, the VGC, the mouse, and so on) in the computer are interrogated. If an internal source generated the interrupt, the interrupt manager calls the appropriate handler.
- If the interrupt wasn't from an internal source, but was from a peripheral card in one of the slots, the computer slows down to the old Apple II speed of 1 MHz, and jumps to the user interrupt vector at location $3FE in Bank $00. When ProDOS first runs, it sets this vector to point to its own internal interrupt manager. The

manager is responsible for finding some way to service the interrupt. This means that every handler associated with a peripheral card should determine whether its card generated the interrupt. The duties of such a handler are discussed later in the chapter.

- Once a handler claims the interrupt and services it, the processor restores the machine state and continues execution from the point where it was interrupted.
- However, if the interrupt is not claimed (and, as a consequence, not serviced), a fatal error occurs. If ProDOS is unable to have the interrupt serviced, it calls a fatal error handler. (In ProDOS 8 this handler would set the screen to 40 columns and display *INSERT SYSTEM DISK AND RESTART—ERR 01*). The user interrupt vector is used mainly by eight-bit data communications programs in servicing interrupts from internal modems or communications cards.

Writing a Handler (Using Blanks)

The Toolbox provides a host of useful functions that make working with interrupts a snap. This section of the chapter will ease you into writing an interrupt handler. The first program example doesn't use interrupts, but it simulates the process of the steps required for real-life interrupt handling.

Actually, this example is quite useful (and fun). The program patches the Apple IIGS's system bell vector with a new beep. After installing this program, the computer will beep with a *fweep* sound reminiscent of a screaming banshee. No more dull, boring *bonk* sound.

The following is the plan of attack for creating the beep.

Setup program. First, start up just the three tool sets: Tool Locator, Miscellaneous Tools, and Memory Manager.

```
        ABSADDR       ON
        KEEP          Beep.Setup
        MCOPY         BeepMacros        ;(use MACGEN to create this file)

Main    START
        phk
        plb                             ;data bank = code bank
        _TLStartUp                      ;start Tool Locator
        _MTStartUp                      ;start Misc Tools
        pha                             ;result space
        _MMStartUp                      ;start Memory Manager
        pla                             ;pull User ID
        sta           UserID
```

Next, call GetNewID to create a new User ID which will be used in allocating a new handle for the beep routine.

```
pha                         ;result space
PUSHWORD     #$A000         ;Type ID / Aux ID
_GetNewID                   ;make an ID
pla
sta          CodeID
```

Then ask NewHandle to allocate a small portion of RAM with the attributes of $C018: It can be any bank or any address, does not need to be page-aligned, and cannot use special memory, cross a bank boundary, or be purged or moved at all.

```
pha                                 ;result space
pha
PUSHLONG     #MBEnd-MyBeep+1        ;Size of block
PUSHWORD     CodeID                 ;CodeID for this handle
PUSHWORD     #$C018                 ;Fixed, locked, bolted down
PUSHLONG     #0                     ;address of the block
_NewHandle
pla                                 ;get handle
plx

sta          0
stx          2
lda          [0]                    ;get long address of block
sta          BlkAddr
ldy          #2
lda          [0],y
sta          BlkAddr+2
```

Once the handle is created and its address determined, place the beep code there by using the BlockMove function. (Yes, the beep routine has to be written as relocatable code. Don't fret. The 65816 has some helpful instructions that make it possible to write relocatable code.)

```
PUSHLONG     #MyBeep                ;Source
PUSHLONG     BlkAddr                ;Destination
PUSHLONG     #MBEnd-MyBeep+1        ;Size
_BlockMove
```

Finally, SetVector is used to patch the beep vector to point to the new beep routine. This program shuts down, and you've finished.

```
                PUSHWORD      #$001B       ;Bell Vector Reference
                PUSHLONG      BlkAddr      ;New Beep Vector Address
                _SetVector

                PUSHWORD      UserID       ;shutdown everything
                _MMShutDown
                _MTShutDown
                _TLShutDown
                rtl

UserID          ds            2
CodeID          ds            2
BlkAddr         ds            4
```

The code that follows is the actual beep routine that is relocated into safe memory. Every time the IIGS is called to beep the speaker, this small routine is called.

```
Speaker     equu    $E0C030     ;speaker toggle location

MyBeep      longa   off
            longi   off
            pha                 ;preserve the registers we munge
            phy
            phx

Fweep0      ldx     #32
Fweep1      ldy     #4
Fweep3      lda     Speaker
            txa
            sec
Wait1       pha
Wait2       sbc     #1
            bne     Wait2

            pla
            sbc     #1
            bne     Wait1

            dey
            bne     Fweep3

            dex
            bne     Fweep1

            plx                 ;restore registers
            ply
            pla
            clc                 ;return with carry clear
MBEnd       rtl

            END
```

Assemble this with *APW* and run the resulting EXE file to install the new beep. (If you're hunting for a way to get the machine to beep at you so you can hear it, pull up the CDA menu and press the space bar or any other illegal key). As long as the computer is turned on, this new beep will be used in place of the old sound.

> Imagine the fun you could have with this if a digitized sound sample were played through the Ensoniq chip, rather than the all-too-common beep.
>
> If you end up liking this new beep better than the *bonk* sound the IIGS normally makes, you can make the process of patching the bell vector part of your ProDOS 16 boot sequence. Just change the file type of the EXE file to TSF ($B7) and copy it to your system disk's SYSTEM/SYSTEM.SETUP directory. It is a TSF (Temporary Startup File), because the entire program doesn't need to be kept in memory. Only the beep portion has to be retained. Every time you boot into ProDOS 16, this new sound will replace the old one, even when you're running ProDOS 8 programs.
>
> Should you wish to go back to using the standard IIGS bell sound, just move the new beep program out of the SYSTEM.SETUP directory and reboot.

This program is an excellent model for getting started on an interrupt installation and servicing program. Some important points need to be made about this program and how it relates to interrupt handlers:

- First, before writing any interrupt handler, consider the programming environment. In the case of this new beep routine, the beep code must be accessible at all times and the code must not be overwritten. That's why a special patch of RAM is allocated by NewHandle explicitly for the beep routine. Since emulation mode programs use banks $00, $01, $E0, and $E1 of the computer, the beep routine could not reside there. The beep code had to be placed outside of special memory. (See Chapter 7, which deals with memory management, for more details).

- The entire installation program is needed only once to install the beep into safe memory and set up the new bell vector. That's why NewHandle is called to allocate space only for the beep handler code. Why waste memory?
- Since NewHandle could end up placing the code anywhere in the machine, the code had to be written so that it didn't use any self-referencing addressing modes. Of course, in this example, that's not a problem. For larger applications, such a program would most likely become a relocatable load segment (more on this and other disk-related matters in Chapter 14).
- The beep routine properly maintains the environment by saving registers before changing them, and then restores them before returning. The handler should avoid modifying any other environment settings (displaying a message on the screen, changing video modes, and so on).

According to the rules, the Apple IIGS's system bell routine is always called in emulation mode with eight-bit registers and must return with the carry clear via an RTL instruction. As with an interrupt handler, there are certain steps to follow to ensure that everything is done correctly.

Recall the sample Applesoft program from the previous section. When run, it caused the computer to ignore interrupts so you couldn't go into the CDA menu after pressing Control–Open Apple–Escape. As soon as interrupt recognition was turned on with the CLI instruction, the CDA menu popped up instantly, without your having to press Control–Open Apple–Escape again. Strange? Not at all.

The reason this happened is because the interrupt of the Keyboard Micro, part of the Apple DeskTop Bus, was still pending and required servicing. The interrupt request line on the CPU was like a telephone that kept ringing until it was finally answered by the 65816 microprocessor. Once interrupt recognition was reestablished, the processor discovered the interrupt was pending and went out to find a way to service it. That's why the CDA menu seemed to come up all on its own. You might chalk it up to delayed reflexes.

Interrupt Vectors

The Beep.Setup program in the last section introduces the Miscellaneous tool set's SetVector function:

Function: $1003
Name: SetVector
Installs an interrupt vector address
Push: Vector reference number (W); Address of routine (L)
Pull: Nothing
Errors: None
Comments: This installs the vector address, but not the interrupt service routine itself.

SetVector is used to change a multitude of system vectors and interrupt handler vectors. The vectors are identified by a unique ID number, as shown in this table:

Reference ID	**Vector Description**
$0000	Tool locator (primary)
$0001	Tool locator (secondary)
$0002	User's tool locator (primary)
$0003	User's tool locator (secondary)
$0004	Interrupt manager
$0005	Coprocessor (COP) manager
$0006	Abort manager
$0007	System death manager
$0008	*AppleTalk* interrupt handler
$0009	Serial communications controller interrupt handler
$000A	Scan line interrupt handler
$000B	Sound interrupt handler
$000C	Vertical blanking interrupt handler
$000D	Mouse interrupt handler
$000E	Quarter-second interrupt handler
$000F	Keyboard interrupt handler
$0010	ADB-response-byte interrupt handler
$0011	ADB-SRQ interrupt handler
$0012	Desk accessory manager (Control–Open Apple–Escape)
$0013	Keyboard-flush-buffer handler (Open Apple–Delete)
$0014	Keyboard-micro interrupt handler
$0015	One-second interrupt handler
$0016	External-VGC interrupt handler
$0017	Other unspecified interrupt handler
$0018	Cursor-update handler
$0019	Increment-busy-flag routine
$001A	Decrement-busy-flag routine

$001B	Bell vector
$001C	BRK vector
$001D	Trace vector
$001E	Step vector
$001F–$0027	Reserved
$0028	Control-Y vector
$0029	Reserved
$002A	ProDOS 16-MLI vector
$002B	Operating system vector
$002C	Message-pointer vector

The actual locations in memory where the vector addresses are stored are presented in Appendix B of *Mastering the Apple IIGS Toolbox.*

SetVector's function is to install the address of a new system or interrupt handler. This is superior to the old global page scheme, where any program had access to all of the system's vectors and could destroy them accidentally. Also, using a tool to set vector addresses means that changes in vector storage locations in later ROM revisions will never be a problem.

SetVector's partner is GetVector. GetVector is used to retrieve the long address of a system/interrupt handler.

Function: $1103
Name: GetVector
Returns the address of an interrupt vector
Push: Result Space (L); vector reference number (W)
Pull: Vector's address (L)
Errors: None

Patching out a vector that will be used only momentarily requires the use of both of these Miscellaneous tool set functions. For example, the following routine demonstrates how you get the current vector address for the monitor's Control-Y vector, patch it out, and then restore it:

```
SetIt       pushlong      #0              ;push long result space
            pushword      #$0028          ;Vector ID = Control-Y vector
            _GetVector                    ;retrieve the current address
            pulllong      OldCtrlY        ;save it for later
            pushword      #$0028          ;Vector ID = Control-Y vector
            pushlong      #NewVect        ;new Control-Y handler address
            _SetVector                    ;set it
            rts
```

```
* Your program then does whatever it must do with the new
* Control-Y vector installed. Before your program quits,
* it restores the old vector address like so. . .

UnSetIt     pushword        #$0028
            pushlong        OldCtrlY
            _SetVector
            rts

OldCtrlY    ds              4                   ;long storage for old Ctrl-Y address
```

GetVector and SetVector can also be used to hook into an existing handler without actually replacing it. For example, if you wanted to have the keyboard-flush handler play a digitized sound sample of a toilet flushing, but still flush the keyboard's type-ahead buffer, you'd proceed as follows:

- Installation
  - Get the keyboard-flush handler address with GetVector.
  - Set the keyboard-flush handler vector with your own routine's address using SetVector.
- Handler operation
  - When the user presses Open Apple–Delete to flush the keyboard buffer, your handler first plays your sound sample.
  - Then it jumps to the original keyboard-flush handler address (the address obtained by the GetVector call in the installation of your handler).

Interrupts in ProDOS 16

SetVector is one way to install an interrupt handler. You can also set one up by going through the operating system, ProDOS 16, if you prefer. This is done mainly for handlers that service interrupts from hardware installed in one of the seven peripheral slots in the IIGS.

Normally, patching into the firmware vectors with SetVector is desired because less overhead is involved since the operating system is bypassed. But the firmware vectors only support those interrupts indigenous to the circuitry in the IIGS and do not make provisions for interrupts from peripheral cards. For these, you have to go through ProDOS 16.

To install an interrupt with ProDOS 16, your program would use the ALLOC_INTERRUPT ProDOS 16 function (number $0031):

```
_ALLOC_INTERRUPT   IParms   ;Allocate the interrupt
bcs                Error    ;branch if error
```

To remove the interrupt allocation in ProDOS, the DEALLOC_INTERRUPT function is used (number $0032):

```
_DEALLOC_INTERRUPT   IParms
bcs                  Error
```

The parameter table for these calls consists of a word and a long word:

```
IParms     anop
int_num    ds      2                  ;this value is returned by ProDOS
int_code   dc      i4'TheHandler'     ;the address of the handler
```

Offset	Size	Description
+$00	word	int_num: Interrupt handler number
+$02	long	int_code: Address of interrupt handler routine

> Actually, only the first parameter is required for DEALLOC_INTERRUPT, but in practice the same parameter block is usually referenced.

When ALLOC_INTERRUPT is used, ProDOS 16 will assign your interrupt handler a unique number which is returned in the first word, int_num. Each time you reference your handler through ProDOS, you use this number (as in the case of memory blocks with the Memory Manager).

Possible error codes returned by these calls are

Error Code	Meaning
$07	ProDOS is busy (it's in the middle of a command already)
$25	Interrupt vector table full (there are already 16 allocated)
$53	Invalid parameter (the handler's address is beyond $FFFFFF)

If ProDOS is busy, you'll have to let it finish what it's doing and then try to allocate the interrupt again later. This is an unlikely event, unless you try to allocate another interrupt and you're already inside an interrupt handler.

Once your interrupt is allocated with ProDOS 16, you can turn on the source of the interrupt and begin handling it. When you wish to deallocate your interrupt, first turn off the interrupt source; then deallocate it.

Environment

When an interrupt handler is called, the computer is placed into a known state, depending on the type of interrupt your handler services and how it is registered with the system. For example, an interrupt handler set up via SetVector can expect the following standard machine configuration:

Code Bank = The bank containing your handler
Data Bank = $00
Emulation = Off (Native mode)
Registers = Eight-bit widths, contents undefined, carry set
Speed = Fast

Your handler returns to the system interrupt manager via RTL.

If your handler is called from the user interrupt vector at $00/03FE, you get the same results as indicated above, except the computer will be running at 1 MHz and emulation mode will be on. Your handler returns to the system interrupt manager via RTS.

If the handler is installed through ProDOS 16, the standard configuration applies, but register widths are set to 16 bits. Your handler returns to ProDOS 16 via RTL.

If your handler modifies any registers or other environmental aspects, it must restore any changes before returning. For example, if you change register widths or their contents, you have to restore them as they were when the handler was initially called. In addition, the carry flag should be cleared before returning if your handler serviced the interrupt. If the carry is set, it indicates to the system that the interrupt was not serviced.

The typical flowchart of an interrupt handler goes something like this:

- Save all the registers and other machine-state information modified in this handler.
- Set up the environment as needed in order to service the interrupt.
- If the handler services an interrupt on a peripheral card, determine whether that card has an interrupt that needs service.
- If it doesn't, set the carry flag and return. Otherwise, service the

interrupt, then clear the interrupt source. (For example, if your handler services one-second clock interrupts, it must reset that interrupt signal before returning. More on this in a later section.)
- Restore the state information saved at the beginning of the handler; then clear the carry flag and return.

Failing to restore the machine state before returning can result in some spectacularly nasty (and possibly fatal) system crashes.

Writing a Handler

Before you can write an interrupt handler, you need to know how to turn on the source that generates interrupts. For peripheral cards in slots 1–7, you'll have to adjust the soft switches mapped to the card's slot. Directions for doing this, and other technical information about the peripheral card, should be found in its manual.

For sources built into the IIGS, the IntSource Miscellaneous tool set function is used to enable or disable interrupts for a particular source. Using it is far easier than messing with softswitches, and it keeps your hands clean, too.

Function: $2303
Name: IntSource
Activates or Deactivates an interrupt source
Push: Source reference number (W) (see below)
Pull: Nothing
Errors: None

Reference Number	Description
$0000	Enable keyboard interrupts
$0001	Disable keyboard interrupts
$0002	Enable vertical blanking interrupts
$0003	Disable vertical blanking interrupts
$0004	Enable quarter-second interrupts
$0005	Disable quarter-second interrupts
$0006	Enable one-second interrupts
$0007	Disable one-second interrupts
$0008	Reserved
$0009	Reserved
$000A	Enable FDB data interrupts
$000B	Disable FDB data interrupts
$000C	Enable scan line interrupts
$000D	Disable scan line interrupts
$000E	Enable external VGC interrupts
$000F	Disable external VGC interrupts

So, to turn on vertical blanking (VBL) interrupts, your program uses

```
pushword    #$0002   ;Enable VBL interrupts
_IntSource
```

To turn VBL interrupts off, use

```
pushword    #$0003   ;Disable VBL interrupts
_IntSource
```

Notice that all the Enable ID numbers are even, and their Disable counterparts are odd. Creative use of equates in your program can make such code self-documenting—for example:

```
Enable     gequu                     0
Dieable    gequ                      1
VBL        gequ                      2
           puehword #Enable+VBL
           _IntSource
           :
           puehword #Dieable+VBL
           _IntSource
```

> Do not attempt to turn on an interrupt source until you've installed the corresponding handler. Doing so is like starting your car while it's in first gear and the clutch is out.

The following complete program listing (Program 12-1) is an actual interrupt installation and handler. Almost as useful as changing the speaker's beep, this program will cycle through all 16 border colors around your screen. Using the one-second interrupt source on the IIGS, the border color will continue to change every second, for a little longer than a minute. It then turns off the one-second interrupts, restores the original interrupt vector, and does its best to clean up memory by unlocking its memory block for purging.

Program 12-1. Second.ASM

```
*------------------------------*

*         Second.ASM           *

*                              *

*   One-Second Interrupt Demo  *

*------------------------------*


        ABSADDR ON

        KEEP    Second
        MCOPY   Second.Mac        ;Create using MACGEN on this file


Main    START

        phk

        plb                       ;data bank = code bank

        _TLStartUp                ;start Tool Locator

        _MTStartUp                ;start Misc Tools

        pha                       ;result space

        _MMStartUp                ;start Memory Manager

        pla                       ;pull User ID

        sta     UserID


        pha                       ;result space

        PUSHWORD #$F000           ;Type ID / Aux ID

        _GetNewID                 ;make an ID

        pla

        sta     CodeID


        pha                       ;result space

        pha

        PUSHLONG #SecEnd-OneSec ;Size of block

        PUSHWORD CodeID           ;CodeID for this handle

        PUSHWORD #$C118           ;Locked, Fixed, (purge=2)
```

```
PUSHLONG #0             ;address of the block
_NewHandle
pla                     ;get handle
plx

sta     0
stx     2
lda     [0]             ;get long address of block
sta     BlkAddr
ldy     #2
lda     [0],y
sta     BlkAddr+2

PUSHLONG #0             ;result space
PUSHWORD #$0015         ;One Second interrupt vector ID
_GetVector
PULLLONG OldVect        ;retrieve old vector address

PUSHLONG #OneSec        ;Source
PUSHLONG BlkAddr        ;Destination
PUSHLONG #SecEnd-OneSec ;Size
_BlockMove              ;move handler code
PUSHWORD #$0015         ;One Second interrupt reference #
PUSHLONG BlkAddr        ;New One Second interrupt handler address
_SetVector

PUSHWORD #$0006         ;Enable 1-sec interrupt Ref Num
_IntSource              ;turn interrupts on

PUSHWORD UserID         ;shutdown everything
_MMShutDown
_MTShutDown
_TLShutDown
_QUIT   QParms
```

```
UserID  ds      2

BlkAddr ds      4

QParms  dc      i4'0'           ;ProDOS 16 Quit Code parameters

        dc      i'$0000'


*------------------------------*

*    Interrupt Handler Code    *

*------------------------------*


Border  EQU     $E0C034         ;RTC/Border color register byte
ScanInt EQU     $E0C032         ;Scanline / 1-sec interrupt source


OneSec  LONGA   OFF             ;This is the handler's entry point

        LONGI   OFF

        phb                     ;save what we end up munging

        pha

        phx

        phy


        phk

        plb                     ;data bank = code bank

        rep     #$30            ;16-bit registers

        LONGA   ON

        LONGI   ON

        per     DataSect        ;push address of data section to stack


        sep     #$20            ;accumulator = 8-bits

        LONGA   OFF

        lda     Border          ;Grab border color

        and     #$F0            ;save upper nibble (RTC bits)

        ldy     #Color-DataSect ;store to Color record in data section

        sta     (1,S),Y
        lda     Border
        inc     A               ;increment it (color is lower nibble)
        and     #$0F            ;truncate any wrapping to upper nibble
        ora     (1,S),Y         ;OR with RTC bits
        sta     Border          ;update the border
        rep     #$20            ;accumulator = 16-bits
        LONGA   ON

        ldy     #Cycle-DataSect ;get Cycle record
        lda     (1,S),Y
        dec     A               ;decrement it
        sta     (1,S),Y         ;update counter
        bne     Exit            ;if counter is not zero, exit

* Once we've cycled through the number of border changes specified,
* we turn off one-second interrupts, restore the old vector, and
* unlock this memory block to make it purgeable when needed.

        PUSHWORD #$0007         ;Disable 1-sec interrupts Ref Num
        _IntSource              ;turn 'em off first

        PUSHWORD #$0015         ;Push 1-Sec vector Ref Num
        ldy     #OldVect-DataSect+2
        lda     (1+2,S),Y       ;push high-word of old vector
        pha
        dey
        dey                     ;(index low-word)
        lda     (1+2+2,S),Y     ;push low-word of old vector
        pha
        _SetVector              ;restore old 1-sec interrupt vector

        ldy     #CodeID-DataSect
        lda     (1,S),Y         ;push code ID
        pha
        _HUnLockAll             ;unlock this block
```

```
Exit    pla                         ;pull PC relative value off stack

        sep     #$30                ;8-bit registers
        LONGA   OFF
        LONGI   OFF
        lda     #%00100000          ;clear 1-sec interrupt source
        sta     ScanInt

        ply                         ;restore registers
        plx
        pla
        plb
        clc                         ;interrupt serviced, return
        rtl

DataSect        ANOP
Color   ds      1                   ;Temporary color value workspace
Cycle   dc      i'64'               ;Number of times border color changes
OldVect ds      4                   ;Original 1-sec interrupt handler address
CodeID  ds      2                   ;User-ID of this memory segment
SecEnd  ANOP

        END
```

Installation of the interrupt handler is similar in most respects to the Beep.Setup program listed earlier in this chapter. The only things different are

- The ID attributes for the GetNewID call do not reference a setup routine.
- The NewHandle attributes assign the memory block a purge level of 2. Even though level 3 means most purgeable, it is reserved for use by the system loader. Since the block is locked, it can't be purged until it is unlocked.
- The current vector for one-second interrupts is preserved before it's changed by the SetVector function.
- IntSource is used to turn on one-second interrupts.

Of course, the handler itself is quite different. Here is a breakdown, starting from the top and dissecting it through to the end, of what the handler does:

```
OneSec  LONGA   OFF   ;This is the handler's entry point
        LONGI   OFF
```

Since this routine is called from the firmware interrupt manager, the system will be placed into native mode with eight-bit registers. Thus, the assembler needs to be placed into the same state at the top of the routine by using the LONGA and LONGI directives.

```
   phb          ;save what we end up destroying
   pha
   phx
   phy
```

The data bank, accumulator, and X and Y registers are all changed in this routine, so they must first be saved by pushing their values onto the stack.

```
   phk
   plb           ;data bank = code bank
   rep    #$30   ;16-bit registers
   LONGA  ON
   LONGI  ON
```

Next, the data bank register is set to the code bank register since this routine runs and accesses data in the same bank. It switches in 16-bit registers and tells the assembler to do likewise.

```
   per  DataSect   ;push address of data section to stack
```

This is a new instruction to most 65816 programmers. PER is used to push the program counter (plus an offset) onto the stack for use in accessing portions of a relocated program. By putting the 16-bit runtime address of the program's data section on the stack, stack-relative indirect addressing can be used to access the data. This makes writing relocatable code nearly painless.

Try doing this with the venerable 6502!

```
   sep     #$20              ;accumulator = 8-bits
   LONGA   OFF
   lda     Border            ;Grab border color
   and     #$F0              ;save upper nibble (RTC bits)
   ldy     #Color-DataSect   ;store to Color record in data section
   sta     (1,S),Y
```

```
lda      Border
inc      A                  ;increment it (color is lower nibble)
and      #$0F               ;truncate any wrapping to upper nibble
ora      (1,S),Y            ;OR with RTC bits
sta      Border             ;update the border
rep      #$20               ;accumulator = 16-bits
LONGA    ON
```

This seemingly complicated series of instructions does one simple task: It increments the screen's border color. It starts by going into 8-bit accumulator mode and grabbing the screen's border color register (also shared by the Real Time Clock chip in the upper nibble). The RTC bits are preserved and stored in the Color data byte via stack-relative indirect addressing. The border color register is fetched once again, incremented, and then the lower nibble of the result is logically ORed with the RTC bits. Finally, the new value is stuffed back into the border color register, and the processor goes back to a 16-bit accumulator.

Most of this confusing footwork is due to the RTC bits needing to be preserved while the lower nibble of Border is incremented, all the while using stack-relative addressing.

> Any time a soft switch or $ExCxxx location is accessed, the accumulator should be set to eight bits. This is because the locations in this chunk of memory are mapped to eight-bit addresses.

```
ldy    #Cycle-DataSect   ;get Cycle record
lda    (1,S),Y
dec    A                 ;decrement it
sta    (1,S),Y           ;update counter
bne    Exit              ;if counter is not zero, exit
```

This portion of the routine decrements a counter that keeps track of the number of times the border color changes. As defined in the data section, 64 iterations will pass before the counter reaches 0. When the sixty-fourth cycle is completed, the following shutdown code is executed:

```
PUSHWORD     #$0007    ;Disable 1-sec interrupts Ref Num
_IntSource             ;turn 'em off first
```

First, the source of the one-second interrupt is shut off. This must be done before the vector is restored in case another one-second interrupt occurs in the middle of this (unlikely, but it's better to be safe than reformatted).

```
PUSHWORD   #$0015                  ;Push 1-sec vector Ref Num
ldy        #OldVect-DataSect+2
lda        (1+2,S),Y               ;pueh high word of old vector
pha
dey
dey                                ;(index low word)
lda        (1+2+2,S),Y             ;pueh low word of old vector
pha
_SetVector                         ;reetore old 1 sec interrupt vector
```

The vector is restored to its original setting at this point. Notice how the byte constants in the stack-relative LDAs increase by 2 each time more data is pushed onto the stack. This is because the program counter (plus data offset), initially pushed on the stack with the PER instruction, hikes up the stack each time something new is pushed, and of course, the reference must compensate for that.

```
ldy             #CodeID-DataSect
lda             (1,S),Y             ;push code ID
pha
_HUnLockAll                         ;unlock this block
```

As the last part of the shutdown sequence, the block that envelops this code is unlocked so that it can be purged whenever the Memory Manager needs to use it.

The DisposeHandle or DisposeAll functions shouldn't be used within the block being disposed. The code that follows the block could be reassigned to some other program in the computer, trashing the instructions and crashing the system.

```
Exit   pla      ;pull PC relative value off stack
```

Remember, the 16-bit address of the data section of this program is still sitting on the stack, so it must be pulled off to maintain harmony.

```
sep     #$30         ;8-bit registers
LONGA   OFF
LONGI   OFF
lda     #%00100000   ;clear 1-eec interrupt source
sta     ScanInt
```

Once again, the computer is placed in eight-bit mode when the $ExCxxx space is being accessed. Storing $20 (%00100000) to ScanInt resets the interrupt signal for one-second interrupts. If this is not done, the processor will be beaten by this interrupt source until the signal is cleared. (For fun, you can try leaving this out just to see what happens.)

Also, recall that when the registers were saved at the top of this handler, the machine was in eight-bit mode. That means that only one byte per register is still sitting on the stack.

```
ply        ;restore registers
plx
pla
plb
clc        ;interrupt serviced, return
rtl
```

After all the registers are restored, the carry flag is cleared to indicate that the interrupt was successfully serviced. The routine returns via an RTL instruction with all registers restored and the machine still in native mode with eight-bit register widths, exactly as it was found at the beginning of this routine.

Clearing Interrupt Sources

Part of servicing any interrupt originating from the IIGS's built-in hardware or on a peripheral card is clearing the interrupt-generating signal. This is the only way the hardware knows that someone has taken care of its interrupt. Once reset, the hardware can ready itself for new interrupts later on. If it isn't cleared, the hardware keeps the interrupt line on the microprocessor ringing nonstop.

Note: Resetting an interrupt signal and disabling the source are two very different things. Disabling an interrupt source will turn it off completely, just like pulling the plug on your electric alarm clock. Resetting the interrupt signal, however, is like hitting the snooze button.

Unfortunately, there is no Toolbox function for clearing the built-in interrupt sources on the IIGS. Perhaps a future version of the Miscellaneous tool set will provide such a handy feature.

For now, your interrupt handler will have to access the hardware register area of the IIGS directly to reset interrupt signals. An example of this is the program in the previous section. It stores $20 to location $E0C032 (called SCANINT). This register contains two bits that correspond to the clearing of scan line and one-second interrupt signals. Writing a 0 to bit 6 of SCANINT resets one-second interrupts. Writing a 0 to bit 5 resets scan line interrupts. The other six bits are unused and should always be set to 0 in writing to SCANINT.

The following table identifies the interrupt reset locations in the Apple IIGS softswitch register area:

Address	**Name**	**Description**
$E0C032	SCANINT	Zero bit 6 to reset one-second interrupts; Zero bit 5 to reset scan line interrupts
$E0C047	CLRVBLINT	Write to clear vertical-blanking (VBL) and quarter-second interrupts
$E0C048	CLRXYINT	Write to clear mouse interrupts

Interrupts from other sources such as serial ports can be cleared by fetching or storing data through the hardware's associated data registers.

The Loch Ness Keyboard Interrupt

One myth about keyboard interrupts is just that: keyboard interrupts. They're a myth in and of themselves. They don't fully exist on the IIGS. The Apple IIGS keyboard really cannot generate an interrupt if, say, you press the M key. Some of the key sequences can cause interrupts, though, such as Control–Open Apple–Escape. But honest-to-goodness data interrupts from keypresses are mythical.

At the moment, keypress interrupts are simulated by some trickery built into the Apple IIGS toolbox. In essence, when IntSource is used to turn on keyboard interrupts, a special task is invoked which runs in the background every 1/60 second. This task looks at the keyboard to see whether a key was pressed, and if it was, jumps to the keyboard interrupt handler installed via SetVector. Why go about it in such a sneaky way?

Unlike most modern computers, which have keyboards that generate true interrupts from keypresses, the Apple IIGS was designed with the opinion that the extra bit of circuitry needed for true interrupts could be sacrificed. But the IIGS's tools development team at Apple designed the Toolbox in such a way as to make a future upgrade of the hardware transparent to software. If a real interrupt-generating keyboard is available for the Apple IIGS someday, all programs that use SetVector and IntSource to establish keyboard interrupts will work just fine, and nobody will be the wiser (except you).

In a HeartBeat

Another form of task processing on the IIGS is provided by the HeartBeat Task Manager, part of the Miscellaneous tool set. These routines allow you to add a series of tasks to perform at any number of 60Hz cycles.

> The HeartBeat Task Manager uses the vertical-blanking interrupt source, which interrupts every 1/60 second.

A HeartBeat task is a routine, usually short, that begins with a special header identifying it as a HeartBeat task. The structure of this header consists of three fields, as shown in this example:

```
TaskHdr     anop
TaskChain   dc     i4'0'        ;pointer to next task
TaskCount   ds     i'60'        ;number of 60Hz cycles before task is run
TaskSig     dc     i'$A55A'     ;special task signature
```

The TaskChain field starts out as a long value of 0. The HeartBeat manager will change this to point to the next task in the HeartBeat task queue, should another be added later.

The TaskCount word is a counter that is decremented by the HeartBeat manager every time the VBL interrupt occurs (every 1/60 second). When this counter reaches 0, your task is executed. It's up to the task to reset the counter to the appropriate number of cycles before returning. Using this method, a task can run from once every 1/60 second to once every 19 minutes.

Finally, the TaskSig word is a constant value of $A55A. If this value is not present here, an error code of $0304 (NoTaskSignature) will be returned when an attempt is made to install the task into the HeartBeat task queue.

Immediately following the task header is the code for the task itself. When the task is called, the computer is placed into native mode using 16-bit registers. The task terminates with an RTL instruction, and unlike what happens with normal interrupt handlers, absolutely nothing needs to be preserved and restored before returning. You needn't fiddle with the carry flag, and even the register widths can be left modified without causing problems. Since your task is invoked indirectly by VBL interrupts, you don't even have to reset any interrupt sources.

Indeed, this is the lazy person's way to install timed background tasks. But there are some advantages to having all the nitty-gritty details handled for you. The only disadvantage is a possible latency in execution of your task should there be a number of other tasks in the queue ahead of yours.

Installing a HeartBeat task is simple. It's done by making a call to SetHeartBeat:

Function: $1203
Name: SetHeartBeat
Places a task into the HeartBeat task manager queue
Push: Address of task header (L)
Pull: Nothing
Errors: $0303, Task already in queue
$0304, No task signature (or bad signature)
$0305, Damaged HeartBeat queue

As easy as using SetHeartBeat is for installing a task, the DelHeartBeat function is used to get rid of one:

Function: $1303
Name: DelHeartBeat
Removes a task from the HeartBeat task queue
Push: Address of task header (L)
Pull: Nothing
Errors: $0304, No task signature
$0306, Task not in queue

This chapter would be incomplete without mentioning a third HeartBeat function, ClrHeartBeat. It removes all tasks from the queue. This should never be used by your applications, though.

Function: $1403
Name: ClrHeartBeat
Removes all tasks from the HeartBeat task queue
Push: Nothing
Pull: Nothing
Errors: None
Comments: Don't make this call

Using the program from the previous section as a starting point, Program 12-2 installs a HeartBeat task that cycles through the border colors for about a minute. The task then removes itself gracefully.

Program 12-2. HeartBeat.ASM

```
*------------------------------*
*        HeartBeat.ASM         *
*                              *
*   One-Second Interrupt Demo  *
*    Using A HeartBeat Task.   *
*------------------------------*


        ABSADDR ON
        KEEP    HeartBeat
        MCOPY   HB.Mac         ;create this file using MACGEN


Main    START
        phk
        plb                    ;data bank = code bank
        _TLStartUp             ;start Tool Locator
        _MTStartUp             ;start Misc Tools
        pha                    ;result space
        _MMStartUp             ;start Memory Manager
        pla                    ;pull User ID
        sta     UserID
```

```
pha                          ;result space
PUSHWORD #$F000              ;Type ID / Aux ID
_GetNewID                    ;make an ID
pla
sta     CodeID

pha                          ;result space
pha
PUSHLONG #SecEnd-OneSec ;Size of block
PUSHWORD CodeID              ;CodeID for this handle
PUSHWORD #$C118              ;Locked, Fixed, (purge=2)
PUSHLONG #0                  ;address of the block
_NewHandle
pla                          ;get handle
plx

sta     0
stx     2
lda     [0]                  ;get long address of block
sta     BlkAddr
ldy     #2
lda     [0],y
sta     BlkAddr+2

PUSHLONG #OneSec             ;Source
PUSHLONG BlkAddr             ;Destination
PUSHLONG #SecEnd-OneSec ;Size
_BlockMove                   ;move handler code

PUSHLONG BlkAddr             ;Pointer to HeartBeat task
_SetHeartBeat

PUSHWORD #$0002              ;Enable VBL interrupt Ref Num
_IntSource                   ;turn interrupts on
```

```
        PUSHWORD UserID         ;shutdown everything
        _MMShutDown
        _MTShutDown
        _TLShutDown
        _QUIT   QParms

UserID  ds      2
QParms  dc      i4'0'           ;ProDOS 16 Quit Code parameters
        dc      i'$0000'
*------------------------------*
*    Interrupt Handler Code    *
*------------------------------*

Border  EQU     $E0C034         ;RTC/Border color register byte
Beats   EQU     60              ;HeartBeats per color change

*** Here's the task header:

OneSec  ds      4               ;task pointer storage chain
BeatCnt dc      i'Beats'        ;approximately every second
        dc      i'$A55A'        ;HeartBeat task signature

*** Here's the task code:

        LONGA   OFF             ;This is the task's entry mode
        LONGI   OFF
        phk
        plb                     ;data bank = code bank
        rep     #$30            ;16-bit registers
        LONGA   ON
        LONGI   ON
        per     DataSect        ;push address of data section to stack
        sep     #$20            ;accumulator = 8-bits
        LONGA   OFF
        lda     Border          ;Grab border color
        and     #$F0            ;save upper nibble (RTC bits)
        ldy     #Color-DataSect ;store to Color record in data section
        sta     (1,S),Y
        lda     Border
        inc     A               ;increment it (color is lower nibble)
        and     #$0F            ;truncate any wrapping to upper nibble
        ora     (1,S),Y         ;OR with RTC bits
        sta     Border          ;update the border
        rep     #$20            ;accumulator = 8-bit
        LONGA   ON

        ldy     #Cycle-DataSect ;get Cycle record
        lda     (1,S),Y
        dec     A               ;decrement it
        sta     (1,S),Y         ;update counter
        bne     Exit            ;if counter is not zero, exit

* Once we've cycled through the number of border changes specified,
* we turn off VBL interrupts, remove the HeartBeat task, and
* unlock this memory block to make it purgeable when needed.

        PUSHWORD #$0003         ;Disable VBL interrupts Ref Num
        _IntSource              ;turn 'em off first

        ldy     #BlkAddr-DataSect+2     ;push address of task on stack
        lda     (1,S),Y                 ;high-word of address
        pha
        dey
        dey                     ;(index low-word)
        lda     (1+2,S),Y       ;push low-word of address
        pha
        _DelHeartBeat               ;remove this task

        ldy     #CodeID-DataSect
        lda     (1,S),Y             ;push memory block ID
        pha
        _HUnLockAll                 ;unlock this block

Exit    pla                         ;pull PC relative value off stack

        per     BeatCnt             ;Update Beat counter
        ldy     #0
        lda     #Beats              ;reset HeartBeat counter for this task
        sta     (1,S),Y
        pla                         ;pull PC relative value

        rtl                         ;then return

DataSect        ANOP
Color   ds      1                   ;Temporary color value workspace
Cycle   dc      i'64'               ;Number of times border color changes
CodeID  ds      2                   ;User-ID of this memory segment
BlkAddr ds      4                   ;Address of HeartBeat task header
SecEnd  ANOP

        END
```

The following things are new or different in the installation portion:

- No vectors are preserved.
- The task is installed with SetHeartBeat.
- VBL interrupts are turned on.

Simply installing a HeartBeat task won't make it go. The VBL interrupt source must be enabled as well.

The task portion is substantially different from the one-second interrupt handler. First, it starts with a HeartBeat task header. This task is set to execute after every 60 heartbeats, which is approximately one second. Notice that none of the processor registers are saved on the stack. This isn't needed for HeartBeat tasks.

The guts of the routine are pretty much the same: Increment the border color, and see whether 64 border changes have been made. If 64 changes have been made, the VBL interrupt source is switched off, the HeartBeat task is deleted with DelHeartBeat, and the block of memory for this task is unlocked.

Before exiting, the routine resets the task counter to 60 beats. If this isn't done, the task isn't ever called again, but remains in the queue.

Finally, the task returns to the HeartBeat manager via RTL.

Interrupt Caveats

Here are a few important notes to keep in mind while working with interrupts:

- The example programs in this chapter use little or no error checking. The intent was to keep the program listings as simple as possible while presenting the study material. Your programs should rely heavily on error checking after each Toolbox call capable of producing errors.
- ProDOS calls and many Toolbox functions, especially those from disk-based tool sets, shouldn't be called from within an interrupt handler. Those resources might not be available at the time of the call. Instead, Apple recommends that such calls be installed into the Scheduler tool set's task queue. Information on that tool set was not available at the time of this writing.
- Switching off an interrupt source from within an interrupt handler is not a common practice. As in the HeartBeat sample program,

which can run in the background while in another application, turning off VBL interrupts can render the application useless if it depends on them.

- You shouldn't use quarter-second interrupts. These are reserved for use by *AppleTalk*.
- In general, use HeartBeat tasks for most timing-related interrupts. This is advantageous since it allows more than one such task to be present at the same time.
- Interrupt handlers are hard to debug with a runtime debugger. This is because the interrupts are occurring in realtime as you're stepping through the code.
- If, while you're programming an interrupt handler, a test run fails and causes the system to crash, it's a good idea to reboot the computer. There's no telling what has become corrupted in memory.

Chapter Summary

Five Miscellaneous tool set functions are presented in this chapter:

- SetVector
- GetVector
- SetHeartBeat
- DelHeartBeat
- ClrHeartBeat

Their official descriptions, including stack parameters and error codes, are discussed within the text of this chapter.

Chapter 13

Desk Accessories

According to the Apple Human Interface Guidelines, a desk accessory is a small program that can be opened while another program is running. Good examples of desk accessories are calculators, note pads, graphic scrapbooks, alarm clocks, utilities, and games. Just about anything found on your typical

(real) desktop is considered a desk accessory.

In the Guidelines, Apple warns that desk accessories should never be too complicated. Some so-called desk accessories for the Macintosh are complete programs unto themselves: spreadsheets, word processors, and graphics programs. They go beyond the limits of desk accessories. Whether they are New or Classic, desk accessories should be quick, efficient, and helpful, short programs that make using the DeskTop interface more practical and enjoyable.

This chapter is about desk accessories. It would be silly to describe desk accessories in detail here, as if this were an introduction to the Apple IIGS. However, desk accessories are a common feature of the IIGS and Macintosh computers. They're just handy, memory-resident programs which are almost always available for use. Everything from the ever-familiar Control Panel to a modeless dialog box/alarm clock can be a desk accessory.

Tell It to the DA

When ProDOS 16 is booted, the desk accessories stored in the SYSTEM/DESK.ACCS subdirectory are installed into memory (see Chapter 3). There can be two types of desk accessories; the advantages of each will be discussed here briefly. The first type is a Classic Desk Accessory (or CDA). This type is available at all times after ProDOS 16 is booted. Classic Desk Accessories can be chosen from the CDA menu by pressing Control–Open Apple–Escape. For example, the Control Panel (where you set your various Apple IIGS options) is merely a Classic Desk Accessory, with the exception that it's part of your ROM and isn't loaded from disk.

A New Desk Accessory (NDA) is only available to programs taking advantage of the DeskTop. NDAs are found in the Apple Menu in DeskTop applications where NDAs are specified. The FixAppleMenu ($1E05) function in the Menu Manager installs NDAs.

The key difference between CDAs and NDAs is that CDAs are always accessible via Control–Open Apple–Escape, and NDAs can only be accessed by DeskTop applications that install them. Otherwise, all desk accessories stay resident in memory until you turn off the computer, reset by pressing Control–Open Apple–Reset, or run the ROM diagnostics by pressing Control–Open Apple–Option–Reset.

It's amusing how Apple has adopted this naming convention of *New* and *Classic* desk accessories. It's suspiciously similar to the Coca-Cola Company's marketing campaign which introduced a new formula for Coke a few years ago in order to compete more successfully with Pepsi (which, as you will recall, a majority of people preferred in blind taste tests). After announcing the New Coke, they dubbed the original concoction *Coca-Cola Classic*. This is of particular interest because Apple Chairman John Sculley was lured away from PepsiCo (the people who produce Pepsi Cola) to work for Apple Computer. Just a coincidence? Apple claims it is.

Since desk accessories are memory-resident, they're usually written in machine language to make them as compact as possible. In fact, because of the structure of desk accessories, it's almost impossible to write them in a high-level language unless the compiler has special provisions for developing them.

Some high-level language compilers do make special allowances for desk accessories. The *TML Pascal* system has a special directive that places the desk accessory header information at the front of your Pascal code. This way, most of the information is handled by the compiler, and your job is simply to write the desk accessory.

Writing a desk accessory is just like writing a normal program. In fact, just about any ordinary program can be turned into a desk accessory simply by adding a bit of extra information and changing the filetype to $B8 for an NDA or $B9 for a CDA. (Note that the extra information is what's important. Simply changing a filetype does not make a desk accessory.)

The steps to creating your own desk accessory differ only in the type of desk accessory you're writing. The following sections of this chapter detail the processes of creating a Classic Desk Accessory and then a New Desk Accessory.

Classic Desk Accessories (CDA)

Of the two types of desk accessories, the Classic Desk Accessory is simpler to program. CDAs are easy to create for two reasons. The first is that they are text-oriented. CDAs pop up on the familiar old

40- or 80-column text screen. You don't have to worry about graphics. The second reason is that they usually don't rely on DOS. Because CDAs can be used at any time, regardless of which operating system is running (ProDOS 8 or 16, DOS 3.3, Pascal, CP/M, or no DOS at all), disk-related functions should be avoided.

It should be noted that if your CDA involves disk activity, it needs to make sure the appropriate DOS is in memory. An Apple IIGS can have a CDA in memory and run another operating system. Never assume ProDOS 16 is present when, in fact, a CP/M program could be running.

If your CDA requires disk activity, it should be able to identify the current DOS environment and inform the user if it's unable to operate.

A Classic Desk Accessory begins with a special header. The header is basically text string information and pointers. For a Classic accessory, the header consists of a title and two long-word pointers:

```
MyCDA   str   "CDA Title"    ;name of DA in the CDA menu
        dc    i4'StartUp'    ;pointer to startup routine
        dc    i4'CleanUp'    ;pointer to a clean up routine
```

It may strike you as odd that this program begins with a text string. If you're sitting there wondering how the CDA can run, remember that CDA files have a special filetype that lets ProDOS know how to load and run them. For Classic Desk Accessories, a filetype of $B9 is used. Disk directories show this as a CDA filetype.

The CDA's title is a standard Pascal string (beginning with a count byte that tells the length of the string). Though it can be as many as 32 characters long, the title should be as short as possible while still being descriptive.

The long pointer to the StartUp routine is actually the address where the CDA code (program) begins. The routine at StartUp is called in full native (16-bit) mode, and it must preserve both the stack pointer (SP) and the data bank register (DBR). The routine must end with a long return (RTL). Those are the only rules to follow. The essence of the CDA resides in this routine.

The long pointer to the CleanUp code contains the address of a routine used to clean house. Whenever the DeskShutDown function is performed, this routine is called. This happens whenever

ProDOS 16 switches to ProDOS 8, or vice versa, and whenever an application makes the DeskShutDown call.

In practice, the CleanUp subroutine should be used to close files, remove interrupt handlers, and do whatever is needed to clean up any mess the CDA may have made. Like StartUp, this routine returns via an RTL. Even if there is no CleanUp routine required by your CDA, the pointer must point to an RTL instruction.

NUMCONV.CDA

The following is a complete Classic Desk Accessory program. After assembling it, change its filetype to $B9 and copy it to your ProDOS 16 disk's SYSTEM/DESK.ACCS directory. To install it, just reboot. Then, when you need a handy hex or decimal number converter, it's only as far away as Control–Open Apple–Escape.

It might be added that this program is somewhat limited in its capabilities. Astute IIGS programmers will find ways to fix up this code or to use it as a skeleton for their own CDAs.

Program 13-1. Number Converter CDA

```
*-----------------------------*

*      Number Converter       *

* Classic Desk Accessory Demo *

*-----------------------------*


       ABSADDR ON

       KEEP    NumConv.CDA

       MCOPY   NumConv.MAC     ;Create this file with MACGEN


NumConv START
       str     'Number Converter'      ;Name of CDA in menu

       dc      i4'StartUp'             ;Pointer to starting routine

       dc      i4'CleanUp'             ;Pointer to clean up routine


StartUp ANOP

       phb                     ;save data bank

       phk                     ;now make data bank = code bank

       plb
```

```
        _TextReset              ;initialize text I/O

        pushlong #Title         ;draw title
        _WriteCString

Loop    pushlong #Prompt        ;prompt for input
        _WriteCString

        pha                     ;result space
        pushlong #InBuf         ;pointer to input buffer
        pushword #16            ;number of characters max to read
        pushword #$8D           ;Return key (MSB set) is EOL character
        pushword #1             ;Echo input
        _ReadLine               ;get the line
        pullword Count          ;get character count
        beq     Exit            ;if equal to zero, exit

        ldx     #0              ;assume hex
        lda     InBuf           ;check for hex
        and     #$7f
        cmp     #'$'
        beq     Conv            ;convert it

        ldx     #2              ;index decimal converter
Conv    jsr     (ProcTbl,X)     ;call either Hex2Dec or Dec2Hex
        bcs     Loop            ;probably string overflow

        sta     NType           ;change result prefix
        pushlong #Result        ;point to string to print
        _WriteCString
        bra     Loop            ;go back for more

Exit    plb                     ;restore bank
```

```
CleanUp ANOP                    ;No clean up needed here

        rtl                     ;return to CDA menu here
Hex2Dec pushlong #0             ;result space

        pushlong #InBuf+1       ;Point to string (skipping $)

        lda     Count

        dec     A               ;length is Count minus 1 (the $)

        pha                     ;length of string

        _Hex2Long               ;long is on the stack now

        pushlong #OutBuf        ;point to output buffer

        pushword #10            ;output string length

        pushword #0             ;unsigned

        _Long2Dec

        lda     #$A0A0          ;two spaces

        rts


Dec2Hex pushlong #0             ;result space

        pushlong #InBuf         ;point to string

        pushword Count          ;number of chars

        pushword #0             ;unsigned

        _Dec2Long               ;Long value is on stack

        pushlong #OutBuf        ;point to output buffer

        pushword #10            ;output string length

        _Long2Hex

        lda     #$A4A0          ;space / dollar sign

        rts
*------------------------------*

*        Data Section          *

*------------------------------*


ProcTbl dc      i'Hex2Dec'

        dc      i'Dec2Hex'


Count   ds      2

InBuf   ds      16
```

```
Result  dc      4c' ',c'-->'

NType   ds      2

OutBuf  dc      10c' ',i1'13,0'


Title   dc      i1'12,17,15',9c' '

        dc      c'|| Number Converter ||'

        dc      9c' ',i1'14,13,13'

        dc      7c' ',c'Press RETURN alone to quit',i1'13'

        dc      7c' ',c'(Start hex numbers with $)',i1'13,13,0'


Prompt  dc      i1'13',c'Number: ',i1'0'
        END
```

If you plan to make extensive use of this desk accessory, you're advised to write a custom input routine. The ReadLine tool is adequate for getting a line of input, but doesn't allow any editing capabilities. For example, if you enter a mistake, pressing the backspace key (←) will not erase the mistake. It will insert an ASCII 8 into the input stream, causing the result of the conversion to be invalid.

New Desk Accessories (NDA)

The formula for producing New Desk Accessories involves more ingredients than the Classic formula requires. Keep in mind the differences between the environment of the NDA and the environment of the CDA. For example, because you're in the DeskTop, it's expected that your NDA will use some form of DeskTop convention. This step alone results in a higher level of programming difficulty than that of creating the CDA.

NDAs are accessible whenever a ProDOS 16 DeskTop application is running in the super-hi-res 320 or 640 mode and the Apple menu is installed into the system menu bar. When run, the NDA can assume that these tool sets are active and started up:

- QuickDraw
- Event Manager
- Window Manager
- Control Manager

- Menu Manager
- LineEdit
- Dialog Manager
- Scrap Manager

Of course, the Tool Locator, the Miscellaneous tool set, the Memory Manager, and other ROM-based tool sets are also available and do not require starting up or shutting down.

> No direct page space is allotted to the NDA, so it must be obtained by calling the Memory Manager's NewHandle function or by tricky use of the stack. The Magnifier program near the end of the chapter contains an example of this.

Like the Classic Desk Accessory, the NDA begins with a special header. (Also like its Classic counterpart, an NDA file can't be run directly, so it's assigned a filetype of $B8, shown as NDA in directory listings.)

The NDA header contains seven fields:

```
MyNDA   dc    i4'OpenNDA'                ;Open the NDA routine address
        dc    i4'CloseNDA'               ;Close the NDA routine address
        dc    i4'TheAction'              ;Do the NDA action routine
        dc    i4'InitNDA'                ;Init NDA StartUp or ShutDown
        dc    i'$0000'                   ;HeartBeat counter
        dc    i'$ffff'                   ;Event mask
        dc    c'--NDA Name\H',3i1'0'     ;NDA item name in Apple menu
```

The first four fields are pointers to subroutines. Each of these special routines is called by the Desk Manager as needed. They must preserve the stack and data bank registers, and end in RTL instructions. In addition, they must preserve the current GrafPort if it is swapped out. Each routine is described in more detail below.

The word value following the pointers is like a HeartBeat counter. (See Chapter 12 for details on HeartBeat tasks and interrupts.) Its value determines the number of 60Hz cycles that will pass before the NDA's Action routine is called with the Run code (more on this below). If this value is 0, the Action routine is called every pass (actually, every time the TaskMaster loop is executed in the DeskTop application). Unlike a HeartBeat task, the NDA does not need to reset this counter after each pass.

Next, a word containing an event mask is used to specify the types of events that the NDA can handle as they relate to actions concerning the NDA. The bits in this word correspond to TaskMaster Event Codes introduced in Chapter 12 of *Mastering the Apple IIGS Toolbox.*

The last field contains a text string in the format of a Menu Manager menu item line. It begins with any two characters, followed by the title of the NDA. The item line is terminated by \H and three zeros. The first two zeros are filled in by the Menu Manager with the item's ID number. The last zero is just a normal C-string terminating character.

The four special routines are described next. For real-life examples of these procedures, see Program 13-2.

The NDA Open Routine is called by the Desk Manager when it wants the NDA to create its window. In fact, the Desk Manager expects the Open routine to return a window port pointer on the stack, and provides result space for it. This is perhaps the trickiest of the four special routines, because the Open subroutine has to modify result space on the stack with the window pointer information. (You'll have a good feel for how result space is changed to meaningful values by the Toolbox after dabbling with this function.) The example NDA program below demonstrates this hair-raising procedure.

After the window is open, the Open routine should set a flag indicating that the window has been created.

The NDA Close Routine is called whenever the close box on your NDA's window is clicked, or whenever the Close menu item (ID = 255) is selected. Your NDA's Close function is used to close the window created by the Open routine. It should test the flag set by the Open routine, then close the window if it's open. Also, it should perform any other housekeeping tasks necessary to close down the NDA gracefully.

The NDA Action Routine is responsible for dispatching a host of handlers to service the events related to the NDA. When called, the Action routine will find a special code in the accumulator which corresponds to the type of action that took place. The nine Action codes are shown in Table 13-1.

Table 13-1. Action Codes

Code	Type	Description
1	EventA	DeskTop event that affects the NDA has taken place. Use the X and Y registers to obtain the address of the event record to further interrogate the event. (X contains the low-order word and Y contains the high-order word of a long address.)
2	Run	It's time to run the guts of the NDA. (See the description of the HeartBeat counter field above.)
3	Cursor	If the NDA window is open, this code is passed to your Action handler each time TaskMaster is called. This is useful for changing the shape of the mouse pointer when it's moved into your NDA's window or some other area on the DeskTop.
4	Menu	A menu item has been selected. The Menu ID and Item ID are passed in the X and Y registers respectively.
5	Undo	Undo selected from the Edit Menu
6	Cut	Cut selected from the Edit Menu
7	Copy	Copy selected from the Edit Menu
8	Paste	Paste selected from the Edit Menu
9	Clear	Clear selected from the Edit Menu

These last five codes correspond to editing functions your NDA may want to handle. If not, the NDA places a zero into the accumulator and returns. Otherwise, the NDA handles the editing action appropriately and returns with a nonzero value in the A register.

The NDA Init Routine is run whenever DeskStartUp or DeskShutDown is called by an application or by the operating system. If DeskStartUp is called, the Init routine will find that the accumulator contains a nonzero value. In this case, the NDA can do whatever it needs to do to prepare itself (usually nothing). If DeskShutDown is called, the Init routine will detect a zero value in the accumulator. It can then clean house as appropriate (for instance, it will close the NDA's window if it's still open).

MAGNIFY.NDA

The following program is an excellent example of a New Desk Accessory. When installed and selected from the Apple menu in a DeskTop program, this NDA will bring up a small window on the screen. It magnifies 512 pixels (a 32 × 16 pixel area), at the mouse pointer's location, and draws the enlarged pixel map in its window.

It demonstrates the structure of a simple New Desk Accessory.

This will run in 640 and 320 modes, though the aspect ratio for 320 mode is a bit out of proportion (the window appears twice as wide as in 640 mode). The budding programmer will want to keep the different screen resolutions in mind when creating an NDA.

Program 13-2. Magnifier NDA

```
*------------------------------*
*          Magnifier           *
*    New Desk Accessory Demo   *
*------------------------------*


        ABSADDR ON
        KEEP    Magnify
        MCOPY   Magnify.MAC             ;Create using MACGEN


Magnify START
        dc      i4'OpenNDA'             ;Open the NDA
        dc      i4'CloseNDA'            ;Close the NDA
        dc      i4'TheAction'           ;Do the NDA action
        dc      i4'InitNDA'             ;Init NDA StartUp or ShutDown
        dc      i'$0000'                ;Heartbeat counter: 0 = each beat
        dc      i'$ffff'                ;Event mask: $ffff = all events
        dc      c'--Magnifier\H',3i1'0' ;NDA item name in Apple menu


*------------------------------*
*   Open the NDA (if closed)   *
*------------------------------*


OpenNDA ANOP
        phb                     ;save data bank (+1 byte to stack)
        phk                     ;data bank = code bank
        plb
```

```
         bit      OpenFlag         ;Have we already created the window?
         bmi      Opened           ;yes -- so skip this stuff

         pha                       ;Result space
         pha
         pushlong #WindowRec
        _NewWindow                ;Create the NDA window
;                                 ;Leave window port pointer on stack
        lda     1,S               ;Get window pointer (low byte) in stack
        sta     WindowPtr         ;Save in memory, and
        sta     4+1+4,S           ;Replace result space on stack

        lda     1+2,S             ;Get window pointer (high byte) in stack
        sta     WindowPtr+2       ;Save in memory, and
        sta     4+1+4+2,S         ;Replace result space on stack
;                                 ;(Recall, WindowPtr is on stack)
        _SetSysWindow             ;Mark this as a system window
        dec     OpenFlag          ;(-1) Flag window as open

Opened  plb
        rtl

*------------------------------*
*   Init NDA StartUp/ShutDown  *
*------------------------------*

; On entry, accumulator is zero if DeskStartUp called.
; Else the A-reg is non-zero if DeskShutDown was called.
InitNDA ANOP
         tax             ;Test accumulator: DeskStartUp called?
         bne     AnRTL   ;yes -- else fall into NDA close routine...
```

```
*----------------------------*
* Close the NDA (if not open) *
*----------------------------*

CloseNDA        ANOP
        phb                     ;save data bank
        phk                     ;data bank = code bank
        plb

        bit     OpenFlag        ;Is the window opened?
        bpl     Closed          ;no -- already closed

        pushlong WindowPtr
        _CloseWindow            ;Close it
        stz     OpenFlag        ;flag it closed, too

Closed  plb                     ;restore data bank
AnRTL   rtl
*----------------------------*
*    Handle NDA Action Event  *
*----------------------------*

TheAction       ANOP
        phb                     ;save data bank
        phy                     ;Save X and Y (address of Event Record)
        phx

        phk                     ;data bank = code bank
        plb
        dec     A               ;(A's range is 1..9, make it 0..8)
        asl     A               ;index procedure table
        tax
        jsr     (ProcTbl,X)     ;Handle the NDA action event
        plx                     ;Restore X...

        ply                     ;...and Y...

        plb                     ;...and DBR

        rtl


NDAUndo         ANOP            ;We don't handle any of these actions

NDACut          ANOP

NDACopy         ANOP

NDAPaste        ANOP

NDAClear        ANOP

                lda     #0      ;flag the above as not handled

NDAMenu         ANOP

NDACursor       ANOP

                rts


NDAEvent ANOP                   ;+2 bytes on stack (return address)

        phd                     ;+2 bytes to stack

        tsc

        tcd                     ;DP = SP (tricky way to get DP access)


        lda     [2+2+1]         ;Grab Event "What" code from stack

        cmp     #9              ;Event codes less than 9 supported here

        bcs     GtEq9           ;=> 9, so skip this


        asl     A               ;index into event procedure table

        tax

        jsr     (EventTbl,X)    ;call the event handler (update only)


GtEq9   pld                     ;Restore direct page

        rts

MouseDown       ANOP

MouseUp         ANOP

KeyDown         ANOP
```

```
AutoKey         ANOP

Activate        ANOP

NotUsed         rts


Update  ANOP


        pushlong WindowPtr

        _BeginUpdate            ;Set VisRgn = Update region


        jsr     NDARun          ;Magnify screen area at mouse location


        pushlong WindowPtr

        _EndUpdate              ;empty update region

        rts


*------------------------------*

*     Main NDA "Run" Event     *

*------------------------------*

NDARun  ANOP

        pushlong #CurrPt        ;get current mouse location

        _GetMouse


        lda     CurrPt          ;compare current and previous points

        cmp     WorkPt

        lda     CurrPt+2        ;(could use EqualPt, but this is faster)

        sbc     WorkPt+2

        bne     Explode         ;if not equal, explode some pixels


        rts                     ;else just return


Explode movelong CurrPt,WorkPt  ;set points


        pushlong WindowPtr
        _StartDrawing           ;Draw in NDA window


        stz     MinY            ;init Y rectangle coordinates

        stz     MaxY

        moveword #16,VCount     ;Explode 16 lines down

        _HideCursor             ;turn off the pointer


VLoop   stz     MinX            ;init rectangle X params
        stz     MaxX

        inc     MaxY            ;adjust Y coord rectangle

        inc     MaxY            ;+2

        moveword CurrX,Xposn    ;reset X posn

        moveword #32,HCount     ;init horiz. counter


HLoop   pha                     ;result space

        pushlong WorkPt

        _GetPixel               ;grab the pixel at current X,Y point

        _SetSolidPenPat         ;set the pen color to it

        add     #3,MinX,MaxX

        pushlong #TheRect

        _PaintRect              ;draw the rectangle

        moveword MaxX,MinX      ;next block over

        inc     XPosn

        dec     HCount

        bne     HLoop


        moveword MaxY,MinY      ;next line down

        inc     YPosn

        dec     VCount

        bne     VLoop

        _ShowCursor             ;turn cursor back on

        movelong CurrPt,WorkPt  ;copy points for next pass comparison

        rts
```

```
*------------------------------*
*         Data Section         *
*------------------------------*

TheRect         ANOP                        ;exploded pixel rectangle
MinY            ds      2
MinX            ds      2
MaxY            ds      2
MaxX            ds      2

VCount          ds      2                   ;explosion counters
HCount          ds      2

WorkPt          ANOP
YPosn           ds      2
XPosn           ds      2

CurrPt          ANOP
CurrY           ds      2                   :current mouse coordinate point
ProcTbl         dc      i'NDAEvent'         ;NDA event handler
                dc      i'NDARun'           ;actual NDA code (guts)
                dc      i'NDACursor'        ;the rest of these are unused
                dc      i'NDAMenu'
                dc      i'NDAUndo'
                dc      i'NDACut'
                dc      i'NDACopy'
                dc      i'NDAPaste'
                dc      i'NDAClear'

EventTbl        dc      i'NotUsed'          ; nothing
                dc      i'MouseDown'        ; Mouse Down
                dc      i'MouseUp'          ; Mouse Up
                dc      i'KeyDown'          ; Key Down
                dc      i'NotUsed'              ; nothing
                dc      i'AutoKey'              ; Auto Key
                dc      i'Update'               ; Update * (only one handled)
                dc      i'NotUsed'              ; nothing
                dc      i'Activate'             ; Activate

OpenFlag        ds      2                       ;Boolean: NDA Window open flag
WindowPtr       ds      4                       ;GrafPortPtr: Window port pointer

wTitle          str     'Magnify'               ;Title of NDA window

WindowRec       dc      i'WRecEnd-WindowRec'    ;Size of parameter table
                dc      i'%1100000010100000'    ;Window frame bits
                dc      i4'wTitle'              ;Pointer to title
                dc      i4'0'                   ;RefCon
                dc      i'0,0,0,0'              ;Zoom rect
                dc      i4'0'                   ;Color table pointer
                dc      i'0,0'                  ;Origin (Y & X)
                dc      i'0,0'                  ;Data area (V & H)
                dc      i'0,0'                  ;Grow box max (V & H)
                dc      i'0,0'                  ;Scroll range (V & H)
                dc      i'0,0'                  ;Paging range (V & H)
                dc      i4'0'                   ;Info bar RefCon
                dc      i'0'                    ;Info bar height
                dc      i4'0'                   ;Definition procedure
                dc      i4'0'                   ;Info bar draw routine
                dc      i4'0'                   ;Content draw routine
                dc      i'40,80,72,176'         ;Position
                dc      i4'-1'                  ;Topmost plane
                dc      i4'0'                   ;Window storage pointer
WRecEnd         ANOP

                END
```

Chapter Summary

All the tools found in the programs in this chapter are discussed in detail in other chapters of this book, as well as in *Mastering the Apple IIGS Toolbox.* None of them deal exclusively with the Desk Manager, however.

Chapter 14

ProDOS

Although this book is about mastering advanced programming techniques for the Toolbox on the Apple IIGS, without ProDOS such a mastery would be nearly impossible. Though they sound like two different beasts, ProDOS and the Toolbox often cross paths. For example, ProDOS is used by the

Standard File Operations tool set, Font Manager, and the Tool Locator. Those tool sets rely upon ProDOS to perform many of their functions.

The Operating System

The *Professional Disk Operating System,* dubbed ProDOS, is little more than a handful of commands to manipulate disk drives. It isn't really an operating system in the classical sense, but it is a smart software interface between an application and a storage device.

Fully detailing the workings and command structure of ProDOS is beyond the scope of this book, so this chapter will have to serve simply as an introduction to ProDOS 16. In it, you will see how to perform a ProDOS command in machine language, C, and Pascal. Included are two lengthy sections that list the ProDOS commands and their parameters. The Standard File Operations tool set is also covered, and a sample program in machine language, C, and Pascal gives you a working example of how ProDOS is used in a real-life situation. Finally, the chapter is wrapped up with a list of ProDOS 16 error codes.

Other Texts

If you're familiar with the way ProDOS 8 or other disk operating systems work, you'll find this chapter a useful reference. But, if you've never worked with file management, it's suggested you check out a programmer's tutorial to working with ProDOS. Some books worthy of mention are

Apple IIGS ProDOS 16 Reference, Apple Computer, 1987. Addison-Wesley.

Beneath Apple ProDOS, Worth and Lechner, 1984. Quality Software.

A note to ProDOS 8 programmers. You're probably familiar with ProDOS 8, the eight-bit version of ProDOS released by Apple Computer in late 1983. ProDOS 8 is the operating system that currently hosts the majority of software for the Apple II series of computers, including such popular programs as *AppleWorks.* But, since ProDOS 8 is geared toward the 64K architecture of earlier Apple IIs, it's inadequate for working with the great expanses of memory and features of the Apple IIGS. ProDOS 16 takes full advantage of the memory you have installed in your computer.

Programmers well versed in the workings of ProDOS 8 will be relieved to know that ProDOS 16 is similar to ProDOS 8 in most respects and is better in many. It's far easier to program than ProDOS 8, even though it's more sophisticated. Function calls are made in a familiar manner, the carry flag indicates that an error occurred, and so forth.

There are many new features along with the basic familiarity. Among other things, parameter tables no longer begin with a count byte. It was the intent of ProDOS 8 to verify the parameter table for a call by making sure the count byte was correct. In a way, this is useless, because the program will probably crash whether the count byte is wrong or the parameter table is referenced incorrectly.

Some of the calls have been renamed, simplified, or have slightly different parameters. Some new calls have been added to make disk operations and file management easier than ever before.

A Call to ProDOS

Before you can use the functions in ProDOS 16, you must first boot a disk formatted and set up for ProDOS 16, such as the System Disk you received when you bought your IIGS. (See Chapter 3 for details on how a ProDOS 16 disk is set up.)

Once loaded, ProDOS 16 can be accessed from machine language by making a long jump to a subroutine at location $E100A8. For example:

```
jsl   $E100A8
```

This address is known as the ProDOS 16 Machine Language Interface (MLI) vector. Calls to the MLI vector are made in full native mode. Your program should preserve the accumulator because ProDOS 16 will store an error result in the A register after each ProDOS call is made. (More on this later.) All other registers are preserved.

A call to ProDOS is followed by two arguments:

- A command number (word)
- A pointer to a parameter list (long)

These arguments are discussed later in this chapter, but, for now, here is a typical ProDOS 16 call:

```
jsl    $E100A8      ;Call the ProDOS 16 MLI
dc     i'$29'       ;$29 = "Quit" command number
dc     i4'QParms'   ;long pointer to parameter list
```

It might appear to be insanely dangerous to use this format for a function call. You would think that after the JSL, the program counter would return to the arguments and careen straight into bit limbo. But in fact, ProDOS 16 will adjust the program counter so that it safely returns to the instruction following the long pointer argument.

This means that you must always call the ProDOS 16 MLI via a JSL instruction, and six bytes of argument information must follow.

Calling ProDOS from Machine Language

As shown in the previous section, calling ProDOS from machine language is done by performing a JSL to $E100A8, followed by two arguments. But the call can be simplified at the source level by using assembler macros. The *APW* Assembler's M16.PRODOS macro file contains macro definitions for every ProDOS 16 function.

Like tool calls, ProDOS 16 macros begin with an underscore, followed by the name of the ProDOS command. The argument to the macro is the address of the parameter list. As an example of using macros for doing a ProDOS call, here's the ProDOS 16 Quit function in *APW* assembler format:

```
        _QUIT   QParms      ;ProDOS 16 quit function call
        . . .

;meanwhile, somewhere else in the program:

QParms  dc      i4'0'       ;longword of zero (no chaining)
        dc      i'0'        ;word of zero (no returning)
```

The _QUIT macro actually expands to the equivalent assembly language statements shown here:

```
jsl  $E100A8
dc   i'$29'
dc   i4'QParms'
```

It's obvious that macros can clean up your ProDOS 16 instructions as well as they do for Toolbox calls.

Calling ProDOS from C and Pascal

Even though C and Pascal have their own built-in disk functions as part of their languages, your high-level programs can access ProDOS directly. The advantage is faster, more efficient programs.

The disadvantage is that your programs will be incompatible when ported to other computer environments. However, since your Desk-Top applications perform tool calls and other IIGS-specific operations, it's probably safe to assume that source code compatibility has already been tossed out the window.

> A general note to C programmers: If your programs can avoid using any of the standard C library functions, including C's disk-related commands such as fopen( ), your executable program will be many times smaller.

To make a ProDOS call in C, your program should include the prodos.h header file at the top of the program:

```
#include <prodos.h>
```

This header file contains predefined symbols for error code numbers, parameter list structures, and the ProDOS function call macros.

To perform the ProDOS 16 Quit function in C, the following statement can be used:

```
QUIT( &QParms );
```

Each ProDOS function call in C follows the naming conventions of ProDOS 16: The names of the C functions are the names of the ProDOS 16 commands, and they're always in capital letters.

A ProDOS command in C requires just one argument: the address of the parameter list. The list is usually a structure containing the needed information to perform the call. Don't forget to place the ampersand (&) in front the structure name, or your program will crash.

In order to use ProDOS in *TML Pascal*, include the ProDOS16 unit symbol file in the USES portion of the program:

```
USES QDIntf,
     GSIntf,
     ProDOS16,
     MiscTools;
```

This makes all the ProDOS 16 functions available to your program. However, naming conventions for ProDOS 16 calls in *TML Pascal* are not as consistent. They all begin with "P16" and do not include underscores. The Quit call in *TML Pascal* is

```
P16Parms.chainPath := StringPtr(0);
P16Parms.returnFlag := 0;
P16Quit( P16Parms );
```

Unfortunately, the arguments to the call are not straightforward either. All arguments to ProDOS calls in *TML Pascal* are referenced through a variant record, called P16Parms in the above example, which is of P16ParamBlk type. Before a call can be made, the fields in the parameter list record must be filled. Setting up parameter lists is discussed later.

Checking for Errors

After each ProDOS call, and depending on which language you're using, you can check for errors:

Language	**Check for Errors**
Machine language	Examine the carry flag
C	Check a variable
Pascal	Test the result of a function

In machine language, if the carry flag is set, an error has occurred, and the accumulator will contain an error code number, as you came to expect in Toolbox calls. For example:

```
        jsl  ProDOS16MLI      ;call the ProDOS 16 MLI ($E100A8)
        dc   i'READ_BLOCK'    ;function number
        dc   i4'RBParms'      ;parameter list pointer
        bcc  NoError          ;if carry is clear, no error occurred

        jmp  HandleDiskErr    ;branch to error handler if carry set
NoError ...
```

In C, the _toolErr global variable holds a nonzero value after making a ProDOS 16 call if an error occurred. The value in _toolErr is the ProDOS 16 error code number.

```
READ_BLOCK( &RBParms );     /* Make the ProDOS 16 call */
if ( _toolErr )             /* If an error occurred. . . */
  HandleDiskErr( );         /* . . .handle it. */
```

With *TML Pascal,* a nonzero value returned by the IOResult function indicates that an error occurred. Any positive, nonzero value is a ProDOS 16 error code number.

```
P16ReadBlock( P16Parms );    { Make the ProDOS 16 call }
IF IOResult > 0 THEN         { If an error occurred. . . }
  HandleDiskErr;             { . . .then handle it. }
```

Error codes are provided at the end of the chapter.

ProDOS 16 Functions

Table 14-1 is a list of the function names and numbers supported by ProDOS 16 Version 1.3, along with a short description of each command.

Table 14-1. Functions Supported by ProDOS 16

Housekeeping Functions		
$01	CREATE	Creates new files or directories
$02	DESTROY	Destroys files or empty directories
$04	CHANGE_PATH	Renames a file or directory, or moves its link
$05	SET_FILE_INFO	Sets various attributes to a file
$06	GET_FILE_INFO	Returns the information set by SET_FILE_INFO
$08	VOLUME	Returns information about a disk volume
$09	SET_PREFIX	Sets one of eight possible prefixes
$0A	GET_PREFIX	Gets one of the eight internal prefixes
$0B	CLEAR_BACKUP_BIT	Clears the backup bit on a file
File Access Functions		
$10	OPEN	Opens an existing file for reading or writing
$11	NEWLINE	Specifies the newline character when reading
$12	READ	Reads data from an opened file
$13	WRITE	Writes data to an opened file
$14	CLOSE	Closes any or all opened files
$15	FLUSH	Writes any unwritten data to a file
$16	SET_MARK	Changes the current position in a file
$17	GET_MARK	Returns the current position in a file
$18	SET_EOF	Sets the end-of-file position for a file
$19	GET_EOF	Gets the end-of-file position for a file
$1A	SET_LEVEL	Sets the system file level for subsequent access
$1B	GET_LEVEL	Gets the current system file level
$1C	GET_DIR_ENTRY	Gets information about entries in a directory

Device Functions

$20	GET_DEV_NUM	Gets the device number for a device or volume
$21	GET_LAST_DEV	Gets the last-accessed device number
$22	READ_BLOCK	Reads a 512-byte block from a device into memory
$23	WRITE_BLOCK	Writes 512 bytes from memory to a block device
$24	FORMAT	Formats a device in various DOS formats
$25	ERASE	Erases a formatted device
$2C	D_INFO	Converts a device number to its device name

Environment Functions

$27	GET_NAME	Gets the pathname of the active application
$28	GET_BOOT_VOL	Gets the volume name where PRODOS was launched
$29	QUIT	Exits the current application
$2A	GET_VERSION	Returns the version number of ProDOS 16

Interrupt Control Functions

$31	ALLOC_INTERRUPT	Allocates an interrupt handler with ProDOS
$32	DEALLOC_INTERRUPT	Deallocates an interrupt handler from ProDOS

> Note that the names given here are the official names used by Apple Computer. C programmers can use these names just as they are. To use them in assembler macros, just put an underscore in front (for instance, _ERASE). For *TML Pascal* programmers, prefix each command with the letters P16 and leave out any underscores (for instance, P16GetBootVol).

Did you notice that some function numbers appear to be missing? This isn't a mistake. Apple Computer has intentionally placed "holes" in the ProDOS 16 command table for future enhancements and additions.

Building a Parameter List

Every ProDOS call requires a parameter list in order to pass information between ProDOS and your program. In machine language, the address of the parameter list follows the command number immediately after the JSL $E100A8. In C and Pascal, the argument to each ProDOS 16 function is the address of the corresponding parameter list.

Values in a parameter list consist of the types listed in Table 14-2.

Table 14-2. Values in a Parameter List

Type	**Size**	**Sample Uses**
Constant	Word (2)	A flag, code number, bit field, reference number
Constant	Long (4)	File offset, block number, and so on
Pointer	Long (4)	Address of a pathname string or storage buffer

> Note that unlike ProDOS 8, only word and long-word values are used in parameter lists in ProDOS 16.
>
> A long pointer to a pathname, such as a prefix, the name of a file, device, or volume, is a Pascal-style string: It begins with a count byte. All parameters that reference strings are long pointers to buffers. Never does a parameter in the list contain string data.

The layout of a sample parameter list for the OPEN ($10) function is demonstrated in Table 14-3.

Table 14-3. Sample Parameter List

Size	**Offset**	**Parameter**	**Description**
Word	00–01	ref_num	Reference number
Long	02–05*	pathname	Long pointer to filename string
Long	06–09	io_buffer	Address of I/O buffer

* You must provide information in this field before making the call to ProDOS. The other fields indicate parameters returned by ProDOS. These returned values are stored in the parameter block when the call is complete. In order to make the OPEN call, all you need to do is supply the pathname pointer, the second parameter. After the call is made, ProDOS fills in the ref_num and io_buffer fields. Offsets are always shown in hexadecimal.

An example of a parameter list in use is demonstrated by this subroutine in assembly language. It makes the OPEN call and references the parameter list, OParms:

```
DoOpen      _OPEN   OParms          ;Open the file
            bcc     Okay            ;If carry is clear, the file is open

            jmp     HandleDiskErr   ;Handle the error

Okay        rts                     ;The program then does whatever
```

```
OParme      ANOP                          ;Parameter list for the OPEN function
Oref_num    ds        2                   ;ref_num returned by ProDOS
Opathname   dc        i4'FileName'        ;long pointer to the filename to open
Oio_buf     ds        4                   ;io_buffer addreee returned by ProDOS

FileName    etr       '/SAMPLE/FILE'      ;Name of file to open
```

This same routine in C could be written like this:

```
OpenRec OParms = { 0,                   /* ref_num */
                   "\p/SAMPLE/FILE"     /* pathname */
                   NULL };              /* io_buf */

DoOpen( )
{
  OPEN( &OParme );
  if ( _toolErr )
    HandleDiskErr;
}
```

And, in *TML Pascal*, an equivalent procedure would be

```
PROCEDURE DoOpen;

VAR OParms: P16ParamBlk;
  FileName: String;

BEGIN
  FileName := '/SAMPLE/FILE';
  OParms.pathname2 := @FileName;
  P16Open( OParme );
  IF IOReeult > 0 THEN
    HandleDiskErr;
END;
```

Using the parameter table for the CLOSE function, shown in the next section, see if you can figure out how to close the file opened above by including just a single function call. It's easier than you might think.

ProDOS 16 Parameter Tables

The following tables describe the parameter lists for every ProDOS 16 call:

> If you're programming in a high-level language, check your compiler's manuals or support files for the appropriate names of each field in a parameter list record.

Table 14-4. Parameter Lists for Every ProDOS Call

$01 CREATE

Size	*Offset*	*Parameter*	*Explanation*
Long	00–03*	pathname	Address of pathname to create
Word	04–05*	access	Access bits (that is, read, write, destroy)
Word	06–07*	file_type	Filetype code number ($00–$FF)
Long	08–0B*	aux_type	Auxiliary filetype code ($0000–$FFFF)
Word	0C–0D*	storage_type	Storage classifier ($01–$0D)
Word	0E–0F*	create_date	Date when file was created (usually $0000)
Word	10–11*	create_time	Time when file was created (usually $0000)

$02 DESTROY

Size	*Offset*	*Parameter*	*Explanation*
Long	00–03*	pathname	Address of pathname to delete

$04 CHANGE_PATH

Size	*Offset*	*Parameter*	*Explanation*
Long	00–03*	pathname	Pathname to rename or move
Long	04–07*	new_pathname	New pathname or location

$05 SET_FILE_INFO

Size	*Offset*	*Parameter*	*Explanation*
Long	00–03*	pathname	Address of pathname to get information on
Word	04–05*	access	Access bits
Word	06–07*	file_type	Filetype code number
Long	08–0B*	aux_type	Auxiliary type (or total_blocks if DIR file)
Word	0C–0D*	unused	
Word	0E–0F*	create_date	Date when file was created
Word	10–11*	create_time	Time when file was created
Word	12–13*	mod_date	Date when file was modified
Word	14–15*	mod_time	Time when file was modified

$06 GET_FILE_INFO

Size	*Offset*	*Parameter*	*Explanation*
Long	00–03*	pathname	Address of pathname to get information on
Word	04–05	access	Access bits
Word	06–07	file_type	Filetype code number
Long	08–0B	aux_type	Auxiliary type (or total_blocks if DIR file)

Word	0C–0D	storage_type	Storage classifier
Word	0E–0F	create_date	Date when file was created
Word	10–11	create_time	Time when file was created
Word	12–13	mod_date	Date when file was modified
Word	14–15	mod_time	Time when file was modified
Long	16–19	blocks_used	Blocks in used by this file (or volume)

$08 VOLUME

Size	*Offset*	*Parameter*	*Explanation*
Long	00–03*	dev_name	Name of device to get information on
Long	04–07	vol_name	Address of buffer to store volume name
Long	08–0B	total_blocks	Volume's total capacity in 512-byte blocks
Long	0C–0F	free_blocks	Number of unused blocks on the volume
Word	10–11	file_sys_id	Filesystem ID (identifies disk format)

$09 SET_PREFIX

Size	*Offset*	*Parameter*	*Explanation*
Word	00–01*	prefix_num	Number of the prefix to set ($0000–$0007)
Long	02–05*	prefix	Address of prefix string

$0A GET_PREFIX

Size	*Offset*	*Parameter*	*Explanation*
Word	00–01*	prefix_num	Number of the prefix to get ($0000–$0007)
Long	02–05*	prefix	Address of returned prefix storage buffer

$0B CLEAR_BACKUP_BIT

Size	*Offset*	*Parameter*	*Explanation*
Long	00–03*	pathname	Address of pathname to have its bit cleared

$10 OPEN

Size	*Offset*	*Parameter*	*Explanation*
Word	00–01	ref_num	Opened file's reference number
Long	02–05*	pathname	Address of pathname to open
Long	06–09	io_buf	Address of io_buffer for this file

$11 NEWLINE

Size	*Offset*	*Parameter*	*Explanation*
Word	00–01*	ref_num	Opened file's reference number
Word	02–03*	enable_mask	Logical AND bitmask used against each byte

Word	04–05*	newline_char	The newline character (in lower byte)

$12 READ

Size	*Offset*	*Parameter*	*Explanation*
Word	00–01*	ref_num	Opened file's reference number
Long	02–05*	data_buffer	Address of buffer where data is read into
Long	06–09*	request_count	Number of bytes to read from file
Long	0A–0D	transfer_count	Actual number of bytes read from file

$13 WRITE

Size	*Offset*	*Parameter*	*Explanation*
Word	00–01*	ref_num	Opened file's reference number
Long	02–05*	data_buffer	Address of data to write into the file
Long	06–09*	request_count	Number of bytes to write
Long	0A–0D	transfer_count	Actual number of bytes written

$14 CLOSE

Size	*Offset*	*Parameter*	*Explanation*
Word	00–01*	ref_num	Opened file's reference number

$15 FLUSH

Size	*Offset*	*Parameter*	*Explanation*
Word	00–01*	ref_num	Opened file's reference number

$16 SET_MARK

Size	*Offset*	*Parameter*	*Explanation*
Word	00–01*	ref_num	Opened file's reference number
Long	02–05*	position	How far into the file to seek

$17 GET_MARK

Size	*Offset*	*Parameter*	*Explanation*
Word	00–01*	ref_num	Opened file's reference number
Long	02–05	position	Current position in file

$18 SET_EOF

Size	*Offset*	*Parameter*	*Explanation*
Word	00–01*	ref_num	Opened file's reference number
Long	02–05*	eof	End-of-file position (file size in bytes)

$19 GET_EOF

Size	*Offset*	*Parameter*	*Explanation*
Word	00–01*	ref_num	Opened file's reference number
Long	02–05	eof	End-of-file position (file size in bytes)

$1A SET_LEVEL

Size	*Offset*	*Parameter*	*Explanation*
Word	00–01*	level	New system file level for opens and closes

$1B GET_LEVEL

Size	*Offset*	*Parameter*	*Explanation*
Word	00–01	level	Current system file level

$1C GET_DIR_ENTRY

Size	*Offset*	*Parameter*	*Explanation*
Word	00–01*	ref_num	Open DIR file's reference number
Word	02–03*	reserved	Must be set to $0000
Word	04–05*	base	Positive or negative code for displacement
Word	06–07*	displacement	Entry displacement from current entry
Long	08–0B*	filename	Address of filename buffer
Word	0C–0D	entry_num	Entry number
Word	0E–0F	file_type	Filetype of the returned entry
Long	10–13	eof	End-of-file position (file size in bytes)
Long	14–17	blocks_used	Number of blocks in use by this entry
Word	18–19	create_date	Date file was created
Word	1A–1B	create_time	Time file was created
Word	1C–1D	mod_date	Date file was modified
Word	1E–1F	mod_time	Time file was modified
Word	20–21	access	Access bits
Long	21–24	aux_type	Auxiliary filetype
Word	25–26	file_sys_id	Filesystem ID

$20 GET_DEV_NUM

Size	*Offset*	*Parameter*	*Explanation*
Long	00–03*	dev_name	Address of device name string
Word	04–05	dev_num	The device number of the named device

$21 GET_LAST_DEV

Size	*Offset*	*Parameter*	*Explanation*
Word	00–01	dev_num	Last accessed device number

$22 READ_BLOCK

Size	*Offset*	*Parameter*	*Explanation*
Word	00–01*	dev_num	Device number to read from
Long	02–05*	data_buffer	Address of 512-byte data buffer
Long	06–09*	block_num	Block number to read

$23 WRITE_BLOCK

Size	*Offset*	*Parameter*	*Explanation*
Word	00–01*	dev_num	Device number to write to
Long	02–05*	data_buffer	Address of 512-byte data buffer
Long	06–09*	block_num	Block number to write

$24 FORMAT

Size	*Offset*	*Parameter*	*Explanation*
Long	00–03*	dev_name	Address of device name to format
Long	04–07*	vol_name	Address of the device's new volume name
Word	08–09*	file_sys_id	Filesystem ID code

$25 ERASE

Size	*Offset*	*Parameter*	*Explanation*
Long	00–03*	dev_name	Address of device name to erase
Long	04–07*	vol_name	Address of the device's new volume name
Word	08–09*	file_sys_id	Filesystem ID code

$27 GET_NAME

Size	*Offset*	*Parameter*	*Explanation*
Long	00–03*	data_buffer	Address of application's pathname buffer

$28 GET_BOOT_VOL

Size	*Offset*	*Parameter*	*Explanation*
Long	00–03*	data_buffer	Address of boot volume's name buffer

$29 QUIT

Size	*Offset*	*Parameter*	*Explanation*
Long	00–03*	pathname	Address of pathname to quit to
Word	04–05*	flags	Return and Restart flags in bits 15 & 14

$2A GET_VERSION

Size	*Offset*	*Parameter*	*Explanation*
Word	00–01	version	Major and minor release versions of ProDOS

$2C D_INFO

Size	*Offset*	*Parameter*	*Explanation*
Word	00–01*	dev_num	Device number to convert
Long	02–05*	dev_name	Address of 32-byte device name buffer

$31 ALLOC_INTERRUPT

Size	*Offset*	*Parameter*	*Explanation*
Word	00–01	int_num	Interrupt reference number
Long	02–05*	int_code	Address of interrupt handling routine

$32 DEALLOC_INTERRUPT

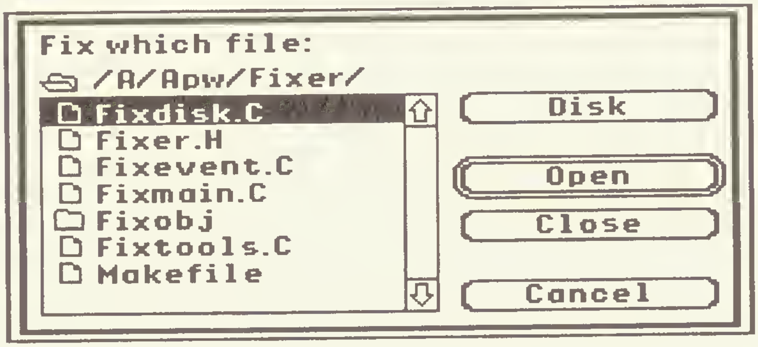
Size	*Offset*	*Parameter*	*Explanation*
Word	00–01*	int_num	Interrupt reference number

* You must provide information in this field before making the call to ProDOS. The other fields indicate parameters returned by ProDOS. These returned values are stored in the parameter block when the call is complete. In order to make the OPEN call, all you need to do is supply the pathname pointer, the second parameter. After the call is made, ProDOS will fill in the ref_num and io_buffer fields. Offsets are always shown in hexadecimal.

Standard File Operations

Tool set 23 ($17), the Standard File Operations tool set, contains a handful of functions that make file selection easier for both the programmer and the user of the program. These tools work in the super-hi-res graphics displays in 320 or 640 modes and present the user with a dialog box containing a list of selectable filenames.

Figure 14-1. A Standard File Operations Dialog Box



Fix which file:

In addition to the standard housekeeping calls (StartUp, Status, and so on) the Standard File Operations tool set provides the functions shown in Table 14-5.

Table 14-5. Functions Provided by Standard File Operations Tool Set

ID	Function	Description
$0917	SFGetFile	Allows the user to select a file to open
$0A17	SFPutFile	Lets the user choose a file to be saved
$0B17	SFPGetFile	Same as SFGetFile, except uses a custom dialog
$0C17	SFPPutFile	Same as SFPutFile, except uses a custom dialog
$0D17	SFAllCaps	Chooses uppercase or mixed case filename displays

Note that these are toolbox calls, not ProDOS 16 commands.

SFGetFile

Use SFGetFile when your program prompts the user to select a file to open. Some examples follow.

In machine language:

```
pushword     #WhereX       ;left coordinate of dialog box
pushword     #WhereY       ;top coordinate of dialog box
pushlong     #Prompt       ;address of prompt string
pushlong     #FilterProc   ;address of filter procedure
pushlong     #TypeList     ;address of valid filetypes list
pushlong     #Reply        ;address of reply record
_SFGetFile
```

In C:

```
SFGetFile( whereX, whereY, "\pPrompt", &filterProc, &typeList, &reply );
```

In Pascal:

```
SFGetFile( WhereX, WhereY, 'Prompt', @FilterProc, @TypeList, Reply );
```

The WhereX and WhereY values specify the position on the screen where the dialog box will be placed.

The Prompt string is a Pascal string which is displayed at the top of the dialog box. This should indicate to the user the purpose of the dialog box by giving a one-line instruction, such as *Select a file to open:*.

Your program may not want the user to be able to select certain files. So you can write your own filter procedure to determine how files are to be displayed in the list. If the address of FilterProc is 0, no filter procedure is called. Otherwise, your filter routine is called for every entry to be placed into the scrolling filename list.

FilterProc is invoked in the following manner by the Standard File tool set:

```
pushword     #0              ;result space that you will fill in
pushlong     #CurrentEntry   ;the address of a directory entry
jsl          YourFilterProc  ;then your filtering routine is called
pullword     ResultCode      ;pull result code
```

As shown, your filter routine must access the two arguments on the stack in order to specify how the current directory entry should be placed in the list. Note that when your filter routine is called, the stack will contain a long return address, followed by a long pointer to a directory entry structure and a word of result space.

The result that your filter routine returns is one of three values:

Value	**Meaning**
0	Do not place the entry into the dialog window
1	Place it in the window, but make it dimmed and not selectable
2	Place it in the window and allow it to be selected

Since the filter procedure must access each file's directory record, you need to know the structure of this 39-byte buffer. This structure is shown in Table 14-6.

Table 14-6. Structure of Directory Record

Offset	**Field**	**Directory Entry Description**
00	storage_type	Storage classifier (upper nibble)
	name_length	Filename length (lower nibble)
01–0F	file_name	String of characters for filename
10	file_type	Filetype code ($00–$FF)
11–12	key_pointer	Pointer to index block
13–14	blocks_used	Number of blocks in use by this entry
15–17	eof	End-of-file position (file's size in bytes)
18–19	create_date	Date file was created
1A–1B	create_time	Time file was created
1C	version	Version of ProDOS that created this file
1D	min_version	Oldest version of ProDOS that can use this file
1E	access	Access bits
1F–20	aux_type	Auxiliary filetype
21–22	mod_date	Date file was last modified
23–24	mod_time	Time file was last modified
25–26	header_pointer	Block number of this file's parent directory

A few fields in this record contain byte values, so you might have to put the processor in eight-bit mode for some operations.

The most straightforward way to filter out a directory entry is done as shown in the following routine. It checks the filetype of the current entry to see how the entry should be displayed:

```
DirEntry  equ       $FC            ;direct page storage for a long pointer

MyFilter  pulllong  Return         ;Pull RTL address off stack
          pulllong  DirEntry       ;set up a long pointer to the entry record
          pla                      ;unload result space from stack for now
          ldy       #$10           ;index into filetype field of entry record
          lda       [DirEntry],Y   ;grab the filetype byte (and next byte)
          and       #$FF           ;make only the filetype byte significant
          ldx       #$00           ;X=0: do not display (assume BAD)
          cmp       #$01           ;is it a BAD block file?
          beq       Done           ;yes

          inx                      ;X=1: display as dimmed (not selectable)
          cmp       #$0D           ;is it a DIR file?
          beq       Done           ;yes

          inx                      ;X=2: display and make selectable
Done      phx                      ;push filter code on stack
          pushlong  Return         ;put return address back on stack
          rtl                      ;and return to it

Return    ds        4              ;Storage for return address
```

Once control returns to SFGetFile, it pulls the filter code off the stack and knows how to handle the entry.

Another way to filter entries is to provide a list of acceptable filetypes by pointing to a TypeList table. Only the file entries which have types listed in the table will be placed into the dialog box. A TypeList begins with a count byte (not a word) followed by a string of byte values indicating valid filetypes. For example:

```
TypeList  dc  i1'4','04,0B,1A,B0'  ;Four document types
```

If you specify a null address for a TypeList, or the list begins with a count byte of 0, this added filtering method is ignored. But, if you specify both a FilterProc and a TypeList, your filter procedure will be called only for the entries that satisfy the file types in the TypeList.

The final argument to the SFGetFile tool call is the address of a Reply record in the format shown in Table 4-6.

Table 4-6. Format of the Reply Record

Offset	Field Name	Reply Record Description
$00–01	good	True if Open clicked; false if Cancel clicked
$02–03	file_type	Filetype code of the file selected
$04–05	aux_file_type	Auxiliary filetype code
$06–15	filename	Name selected from name list (16 bytes)
$16–96	full_pathname	Full pathname to file (129 bytes)

This record is filled in with values by SFGetFile whenever the user clicks the Open or Cancel buttons.

Your program will know whether it should continue with file operations by examining the good field of the Reply record. If it contains a false (0) value, the program knows that the user clicked Cancel. Any nonzero value means the Open button was clicked.

SFGetFile also returns the filetype and aux_file_type codes for the file selected. This information might be useful to your program.

The filename and full_pathname fields are Pascal-style strings. The 15-character filename is the name of the file selected as it was shown in the scrollable list (mixed case and all). The full_pathname is a fully qualified pathname to the file selected.

After a file is chosen, the current ProDOS prefix is set to the subdirectory (or, the folder) that contains the selected file.

SFPutFile

Use the SFPutFile function to allow the user to select a file when saving information to disk. If the user selects a file that already exists, SFPutFile will bring up a second dialog box on its own to ask the user whether it's okay to overwrite the existing file.

In machine language:

```
pushword  #WhereX     ;left edge of dialog
pushword  #WhereY     ;top edge
pushlong  #Prompt     ;address of prompt string
pushlong  #OrigName   ;address of original filename
pushword  #MaxLen     ;maximum number of characters in name
pushlong  #Reply      ;address of reply record
```

In C:

```
SFPutFile( whereX, whereY, "\pPrompt", &origName, maxLen, &reply );
```

In Pascal:

```
SFPutFile( WhereX, WhereY, 'Prompt', @OrigName, MaxLen, Reply );
```

The WhereX and WhereY values specify the position on the screen where the dialog box will be placed.

The Prompt string is a Pascal string and should provide a message, such as *Save document to:*, giving the user an idea of the operation at hand.

OrigName is the address of a Pascal string to be placed into the EditLine item in the SFPutFile dialog. OrigName normally points to the filename returned by SFGetFile when the file was first opened.

MaxLen indicates the number of characters that can be entered in the EditLine item. This is usually 15 since the current implementation of ProDOS limits filenames to 15 characters.

The last argument is the address of the Reply record as described in the SFGetFile section. Both SFGetFile and SFPutFile use the same Reply record format.

SFAllCaps

Normally filenames are shown in mixed case in the scrolling list of names in a Standard File dialog box, but if you prefer all uppercase, use the SFAllCaps function with a Boolean value of true (any nonzero value). A false value (0) indicates mixed case.

In machine language:

```
pushword    #1   ;true: show names in all uppercase
_SFAllCaps
```

In C and Pascal:

```
SFAllCaps( TRUE );
```

A Nonredundant Example

Because ProDOS calls occupy a relatively small portion of the sample program for this chapter, things will be handled a bit differently. Rather than providing three huge programs in machine language, C, and Pascal, only one program is presented in its entirety. C was chosen for the job because it is midway between the low-level control of machine language and the high-level ease of Pascal. The section containing the ProDOS calls is provided in both machine language and Pascal, however.

The program listed below, CRC.C, is a 320-mode desktop program that calculates a cyclic redundancy checksum on the contents of a disk file. Unlike most of the other programs in this book, CRC.C uses no pull-down menus. Instead, the program is centered around the Standard File Operation's SFGetFile dialog box on the desktop. The user selects a file and clicks the Open button to begin the CRC calculation. To quit the program, the user simply clicks on the Cancel button. Putting a pull-down menu into a program like this would introduce an unnecessary step, so menus are left out.

> What in the world is a CRC? A CRC is a calculation on a piece of data that results in a unique 16-bit value. It's used mainly in data communications protocols to ensure the correct transfer of a file over less-than-pristine telephone connections. For everyday purposes, it can be used to quickly compare two files that are supposed to be identical to see if they are different.

CRC.C

This program demonstrates how to use the SFGetFile function to allow the user to select a file from disk. It will open the selected file, read through it, trap the famous "end-of-file" error, and close the file; a typical file-handling procedure. Note also how this program can easily be changed to run in 640 mode just by modifying two definitions near the top of the program.

Program 14-1. CRC.C

```
/*------------------------------------------*
 *                  CRC.C                   *
 *   Cyclic Redundancy Checksum Calculator  *
 *          Written by Morgan Davis         *
 *------------------------------------------*/


#include <types.h>
#include <locator.h>
#include <memory.h>
#include <misctool.h>
#include <quickdraw.h>
#include <qdaux.h>
```

```
#include <event.h>

#include <window.h>

#include <lineedit.h>

#include <control.h>

#include <dialog.h>

#include <stdfile.h>

#include <intmath.h>

#include <prodos.h>


#define Mode        320                 /* Screen Mode (320 or 640) */

#define MasterSCB   mode320             /* MaxWidth mode (320 or 640) */


#define Center      ((Mode - 1) / 2)    /* Center pixel column */

#define BoxWidth    240                 /* Dialog box size & location */

#define BoxHeight   70

#define BoxX        (Center - (BoxWidth / 2))

#define BoxY        40


#define BufferSize  2048        /* Size of file input buffer */


GrafPortPtr DialogPort;         /* Dialog port */

WmTaskRec   EventRec;           /* Event Record Structure */

SFReplyRec  Reply;              /* Standard File Record Structure */

OpenRec     OParms;             /* Open File parameter list */

FileIORec   RParms;             /* Read File parameter list */

QuitRec     QParms = { 0L, 0 }; /* Quit parameter list */


Word        UserID,             /* Our User ID */

            MemID,              /* Memory Management ID */

            CRC;                /* The CRC */
```

```
Word    Toolist[] = { 6,

            14, 0,  /*          Window Manager */

            16, 0,  /* 0x100    Control Manager */

            18, 0,  /*          QuickDraw II Aux */

            20, 0,  /* 0x100    LineEdit */

            21, 0,  /*          Dialog Manager */

            23, 0   /* 0x100    Standard File */

};                  /* 0x300    QuickDraw II */

                    /* 0x100    Event Manager */

                    /* =====    total direct page space is ... */

#define DPageSize       0x700L


char        *DPBase;            /* Direct Page base pointer */


ItemTemplate OKItem = { ok,

                        BoxHeight-22, BoxWidth-68, 0, 0,

                        buttonItem,

                        "\p OK ",

                        0, 0, NULL

};


DialogTemplate CRCBox = {

                        BoxY, BoxX, BoxY+BoxHeight, BoxX+BoxWidth,

                        1,

                        NULL,

                        &OKItem,

                        NULL

};
```

```
/*------------------------------*
 *    Handle Toolbox Errors     *
 *------------------------------*/

ErrChk()                        /* Check for error, die if so */
{
    if (_toolErr) SysFailMgr(_toolErr, NULL);
}

/*------------------------------*
 *  Manage Direct Page Buffers  *
 *------------------------------*/

char    *GetDP(bytes)
Word    bytes;
{
    char *OldDP = DPBase;
    DPBase += bytes;            /* Update base level pointer */
    return (OldDP);             /* Return old DPBase pointer */
}

/*------------------------------*
 *        Start Up Tools        *
 *------------------------------*/

StartUpTools()
{
    Word    GetDP();        /* Force words from GetDP */

    TLStartUp();                                                ErrChk();
    MemID = (UserID = MMStartUp()) | 256;                       ErrChk();
    MTStartUp();                                                  ErrChk();
    DPBase = *(NewHandle(DPageSize, MemID, 0xc005, NULL));        ErrChk();
    QDStartUp(GetDP(0x300), MasterSCB, 0, UserID);                ErrChk();
    EMStartUp(GetDP(0x100), 0x14, 0, Mode, 0, 200, UserID);       ErrChk();

    SetForeColor(9);
    SetBackColor(0);
    MoveTo(20,20);
    DrawCString("One moment...");
    InitCursor();

    LoadTools(Toolist);                     ErrChk();
    QDAuxStartUp();                         ErrChk();
    WindStartUp(UserID);                    ErrChk();
    CtlStartUp(UserID, GetDP(0x100));       ErrChk();
    LEStartUp(UserID, GetDP(0x100));        ErrChk();
    DialogStartUp(UserID);                  ErrChk();
    SFStartUp(UserID, GetDP(0x100));        ErrChk();

    Desktop(5, 0x40000030);
}

/*------------------------------*
 *  Calculate CRC on a Buffer   *
 *------------------------------*/

/* A CRC is the result of a mathematical operation based on the
 * coefficients of a polynomial when multiplied by X^16 then divided by
 * the generator polynomial (X^16 + X^12 + X^5 + 1) using modulo two
 * arithmetic.  That's okay, I don't understand it either.
 */
```

```
CalcCRC (ptr, count)
char    *ptr;           /* Pointer to start of data buffer */
Word    count;          /* Number of bytes to scan through */
{
   Word x;


   do {
       CRC ^= *ptr++ << 8;              /* XOR hi-byte of CRC w/ data */
       for (x = 8; x; --x)              /* Then, for 8 bit shifts...  */
           if (CRC & 0x8000)            /* Test hi order bit of CRC   */
               CRC = CRC << 1 ^ 0x1021; /* if set,shift & XOR w/$1021 */
           else
               CRC <<= 1;               /* Else, just shift left once.*/
   } while (--count);                   /* Do this for all bytes      */
}


/*------------------------------*
 *      Get CRC on the File     *
 *------------------------------*/


GetCRC (pathname)
char    *pathname;      /* Pointer to full pathname */
{
    Word    Error;
    Boolean EOF = FALSE;
    char    Buffer[BufferSize];


    OParms.openPathname = pathname;
    OPEN (&OParms);                             /* Open the file */

    Error = _toolErr;

    if (!Error) {                               /* If no error ... */

        RParms.fileRefNum = OParms.openRefNum;

        RParms.dataBuffer = Buffer;

        RParms.requestCount = BufferSize;

        do {

            READ (&RParms);                     /* ...read some data */

            Error = _toolErr;

            if (Error) {                        /* If error... */

                EOF = TRUE;                     /* flag EOF */

                if (Error == eofEncountered)    /* EOF isn't fatal... */

                    Error = 0;                  /* ...so zero error */

            } else

                CalcCRC(Buffer, RParms.transferCount);

        } while (!EOF);

    }

    CLOSE (&OParms);                    /* Close the file */

    return (Error);                     /* And return error code */

}


/*------------------------------*

 *   Show CRC in Modal Dialog   *

 *------------------------------*/

ShowCRC()

{

    char    *CRCstr = "$xxxx";

    char    *ErrorMsg = "ProDOS Error!  Code ";

    Word    Error;


    WaitCursor();
```

```
DialogPort = GetNewModalDialog(&CRCBox);    /* Create modal dialog */

SetPort(DialogPort);


MoveTo(10,20);

DrawCString("Getting CRC on ");             /* Print a prompt */

DrawString(Reply.filename);

DrawCString("...");

MoveTo(BoxWidth/2 - 26, 36);


CRC = 0;                                /* Init CRC at zero */

Error = GetCRC(Reply.fullPathname);     /* Get CRC on the file */

if (Error) {                            /* If an error occurred... */

    MoveTo(10,36);                      /* ...print a message */

    DrawCString(ErrorMsg);

    SysBeep();

    CRC = Error;
    }


    Int2Hex(CRC, CRCstr + 1, 4);            /* Make CRC printable */

    DrawCString(CRCstr);                    /* Then print it */

    InitCursor();

    ModalDialog(NULL);                      /* Wait for OK button */

    CloseDialog(DialogPort);                /* Close the dialog */

}


/*--------------------------------*

 *      Shutdown Toolsets         *

 *--------------------------------*/


ShutDownTools()

{
    SFShutDown();
    LEShutDown();
    CtlShutDown();
    WindShutDown();
    EMShutDown();
    QDAuxShutDown();
    QDShutDown();
    MTShutDown();
    DisposeAll(MemID);
    MMShutDown(UserID);
    TLShutDown();
}


/*-------------------------------*
 *            Main               *
 *-------------------------------*/


/* Display a Standard File Operations "Get" Dialog and wait for a
 * file to be selected.  If Cancel is selected, the program quits.
 */


main()
{
    StartUpTools();
    do {
        SFGetFile(Center-130, 35, "\pCalculate CRC on:", 0L, 0L, &Reply);
        if (Reply.good)      /* If Open clicked ... */
            ShowCRC();       /* ... do the CRC      */
    } while (Reply.good);
    ShutDownTools();
    QUIT (&QParms);
}
```

CRC.ASM

To complete the machine language version of this program, simply steal parts of the MODEL.ASM and other examples from this book which correspond to most of the routines in CRC.C. Program 14-2 and 14-3 are two new subroutines, GetCRC and CalcCRC in machine language.

Program 14-2. Calculate CRC on a Buffer

```
*-------------------------------*
*  Calculate CRC on a Buffer    *
*-------------------------------*


CalcCRC ldy     #0              ;init index into buffer
NxtByte shortm                  ;go to 8-bit accumulator
        lda     Buffer,Y        ;grab a character
        iny                     ;bump index
        eor     CRC+1           ;fix high-order byte of CRC
        sta     CRC+1
        longm                   ;back to 16-bit accumulator

        ldx     #8              ;init shift counter
Shift   asl     CRC             ;shift CRC left once always
        bcc     Next            ;if bit 15 was clear, skip the XOR

        lda     CRC             ;XOR CRC with $1021
        eor     #$1021
        sta     CRC
Next    dex
        bne     Shift           ;do this 8 times
        dec     xferNum         ;More bytes to do?
        bne     NxtByte         ;yes if count not zero

        rts

CRC     ds      2               ;here's the CRC word
```

Program 14-3. Get CRC on the File

```
*------------------------------*
*   Get CRC on the File        *
*------------------------------*

GetCRC  _OPEN   OParms          ;Open the file
        bcs     StopErr         ;stop if error occurred

        moveword Oref,Rref      ;Copy reference number
RdLoop  _READ   RParms          ;Read data from file into Buffer
        bcs     ChkErr          ;error occurred -- check for EOF

        jsr     CalcCRC         ;no error, so update CRC
        bra     RdLoop          ;and go back to read more until EOF

ChkErr  cmp     #$4C            ;end of file error?
        bne     StopErr         ;no, so return it

        lda     #0              ;flag no error -- EOF isn't fatal
StopErr sta     Error           ;save error return code
         CLOSE  OParms          ;close the file
        rts                     ;and return

Error   ds      2               ;error code location
```

```
Buffer  ds      BufferSize      ;data buffer


OParms  ANOP                    ;OPEN Parameter List
Oref    ds      2               ;open file reference number
        dc      i4'pathname'    ;pointer to pathname
        ds      4               ;address of I/O buffer


RParms  ANOP                    ;READ Parameter List
Rref    ds      2               ;reference number for reading
        dc      i4'Buffer'      ;pointer to data buffer
        dc      i4'BufferSize   ;size of buffer
xferNum ds      4               ;transfer count
```

CRC.PAS

The GetCRC and CalcCRC routines in *TML Pascal* are shown in Program 14-4.

Program 14-4. Calculate CRC on a Buffer

```
{*------------------------------*
 *  Calculate CRC on a Buffer   *
 *------------------------------*}


{ Note that the global variable CRC has a range of $0000..$FFFF }


PROCEDURE CalcCRC (bufPtr: Ptr; count: Integer);
VAR     x:      Integer;
        Data:   $0000..$FFFF;
BEGIN
    REPEAT
        Data := bufPtr^;
        FOR x := 1 TO 8 DO
            Data := BitSL (Data);
        CRC := BitXOR (CRC, Data);

        bufPtr := Pointer (Longint (bufPtr) + 1);

        FOR x := 1 TO 8 DO

            IF CRC > $7FFF THEN

                CRC := BitXOR (BitSL (CRC), $1021)

            ELSE

                CRC := BitSL (CRC);

        Dec (count);

    UNTIL (count = 0);

END;


(*------------------------------*

 *      Get CRC on the File     *

 *------------------------------*)


FUNCTION GetCRC (pathname: StringPtr) : Integer;

VAR Error:  Integer;

    EOF:    Boolean;

    Buffer: Packed Array [0..BufferSize] of Byte;

    OParms: P16ParamBlk;

    RParms: P16ParamBlk;

BEGIN

    EOF := FALSE;

    OParms.Pathname2 := pathname;

    P16Open (OParms);

    Error := IOResult;

    IF Error = 0 THEN BEGIN

        RParms.refNum := OParms.refNum;

        RParms.dataBuffer := @Buffer[0];

        RParms.requestCount := BufferSize;
        REPEAT
            P16Read (RParms);
            Error := IOResult;
            IF Error > 0 THEN
                BEGIN
                    EOF := TRUE;
                    IF Error = $4C THEN Error := 0;
                END
            ELSE
                CalcCRC(@Buffer[0], RParms.transferCount);
        UNTIL EOF;
    END;
    P16Close (OParms);
    GetCRC := Error;
END;
```

Disk Errors

Table 14-7 is a complete list of error codes that can be returned by the ProDOS 16 operating system. (See the "Checking for Errors" section in this chapter for details on how to detect and handle errors).

Table 14-7. ProDOS 16 Error Codes

Number	Meaning
$00	No error
$01	Invalid call number
$07	ProDOS is busy
$10	Device not found
$11	Invalid device request
$25	Interrupt vector table full
$27	I/O error
$28	No device connected
$2B	Disk is write-protected
$2E	Disk switched, files open
$2F	Device not online
$30–$3F	Device-specific errors

Number	Meaning
$40	Invalid Pathname
$42	File control block table full
$43	Invalid reference number
$44	Path not found
$45	Volume not found
$46	File not found
$47	Duplicate pathname
$48	Volume full
$49	Volume directory full
$4A	Version error
$4B	Unsupported storage type
$4C	EOF encountered, out of data
$4D	Position out of range
$4E	Access not permitted
$50	File is open
$51	Directory structure damaged
$52	Unsupported volume type
$53	Invalid parameter
$54	Out of memory
$55	Volume control block full
$57	Duplicate volume
$58	Not a block device
$59	Invalid file level
$5A	Block number out of range
$5B	Illegal pathname change
$5C	Not an executable file
$5D	File system not available
$5E	Cannot deallocate /RAM
$5F	Return stack overflow
$60	Data unavailable

Chapter Summary

The following functions are part of the Standard File Operations tool set, which is presented in this chapter:

Function: $0117
Name: SFBootInit
Initialize the Standard File tool set environment
Push: Nothing
Pull: Nothing
Errors: None
Comments: Applications do not make this call.

Function: $0217
Name: SFStartup
Starts up the Standard File Operations tool set
Push: User ID (W); Direct Page (W)
Pull: Nothing
Errors: None
Comments: Call this before using Standard File functions.

Function: $0317
Name: SFShutdown
Shuts down the Standard File tool set and frees some memory
Push: Nothing
Pull: Nothing
Errors: None
Comments: Call this when your application is done using Standard File Operations.

Function: $0417
Name: SFVersion
Get the current version of the Standard File tool set
Push: Result Space (W)
Pull: Version number (W)
Errors: None

Function: $0517
Name: SFReset
Reset the Standard File Operations tool set
Push: Nothing
Pull: Nothing
Errors: None

Function: $0617
Name: SFStatus
Determine if the Standard File Operations tool set is active
Push: Result Space (W)
Pull: Active flag (W)
Errors: None
Comments: The flag is 0 if false and nonzero if true.

Function: $0917
Name: SFGetFile
Lets the user choose a specific file from a dialog box
Push: X position of dialog box (W); Y position of dialog box (W); Pointer to dialog box title string (L); Pointer to filtering subroutine (L); Pointer to list of valid file types (L); Pointer to returned pathname record structure (L)

Pull: Nothing
Errors: None
Comments: Title string starts with a count byte. Calling of the filtering routine can be inhibited by using $00000000 as its address. The filtering routine should return via RTL. File type list starts with a count byte. Record structure returned is the following: Open Flag (W); File type (W); Auxiliary file type (W); filename (16 bytes); full pathname to file (129 bytes).

Function: $0A17
Name: SFPutFile
Lets the user choose a filename for saving information to disk
Push: X position of dialog box (W); Y position of dialog box (W); Pointer to dialog box title string (L); Pointer to string containing original filename (L); Maximum length of name (W); Pointer to returned pathname record structure (L)
Pull: Nothing
Errors: None
Comments: Returned record structure is the same as SFGetFile.

Function: $0B17
Name: SFPGetFile
Allows user to choose a filename from a custom dialog box
Push: X position of dialog box (W); Y position of dialog box (W); Pointer to title string (L); Pointer to filtering routine (L); Pointer to file type list (L); Pointer to dialog template structure (L); Pointer to modal dialog event handler (L); Pointer to returned pathname record structure (L)
Pull: Nothing
Errors: None
Comments: Same as SFGetFile except for the template pointer and modal dialog activity handler (see Dialog Manager section for details).

Function: $0C17
Name: SFPPutFile
Gives the user a custom dialog box to choose a filename for saving information to disk.
Push: X position of dialog box (W); Y position of dialog box (W); Pointer to dialog box title string (L); Pointer to string containing original filename (L); Maximum length of name (W); Pointer to dialog template structure (L); Pointer to modal dialog event handler (L); Pointer to returned pathname record structure (L)

Pull: Nothing
Errors: None
Comments: See SFPGetFile.

Function: $0D17
Name: SFAllCaps
Sets the case mode for filenames in dialog boxes
Push: Case flag (W)
Pull: Nothing
Errors: None
Comments: If case flag is true (nonzero), all filenames in SFO dialog boxes will be shown without conversion to lowercase.

Appendices

Appendix A

Apple's Human Interface Guidelines

The uniting idea behind the Macintosh and Apple IIGS desktop, windows, menu bars, icons, and dialog boxes is to give all software applications a universal look and feel. Apple wants its computers to be easy to learn and to use. To accomplish this, all software should follow the same conventions and use the same or similar methods of accomplishing many tasks.

> Witness the rabble of MS-DOS software, with its many programs and varying uses of graphics, the keyboard, and other conflicting methods of operating a program. The Human Interface Guidelines provide sanity and order in an operating system that might otherwise be just as confusing as the rest.

Contrary to what you've read, following the guidelines is not called *user-friendly* programming. Instead, Apple refers to it as *user-centered* programming. Most programs are written by programmers who wish to amaze other programmers. As a programmer yourself, you've probably been frustrated with the way things are supposed to be done using the desktop interface. After all, wouldn't it be much easier and faster to type an MS-DOS–like command such as COPY A:*.* C: \ROOT \DEV /B?

Perhaps you have noticed that the user interface of many non-Apple programs is poorly thought out. Among the dozens of word processors available for MS-DOS computers, there are radically different procedures to perform the same tasks. Some word processors have their own conventions and, for the convenience of users, allow alternate keypresses to mimic other word processors. Some even have vastly different sequences of commands within a single program to achieve similar results. This is the sort of disarray that naturally occurs when there is no enforced standard.

Apple has worked on its Human Interface Guidelines for years. The idea behind the guidelines is to make all programs running on

Apple computers behave the same, or enough alike that you only need to learn one technique for accomplishing similar tasks in several programs.

This appendix presents certain ideas and philosophies of the Human Interface Guidelines. It was decided that these ideas should all be placed together here, rather than scattered throughout the book. After all, you are a programmer. And usually the last thing you'll consider is how the first-time user will feel about using your program. Now that you know how to program the Apple IIGS Toolbox, it's time you learned how to present it to the user.

Programs are not judged on speed alone. Many programmers pride themselves in writing fast, compact code. Getting the job done, and done quickly, is important. But magazine reviewers and software salespeople will not recommend programs that don't follow these guidelines.

> Reading the Human Interface Guidelines is like reading a dog-eared, highlighted college text. The list is full of interesting ideas, thoughts, and reasons explaining why Apple did what it did in designing the Macintosh/Human interface.
>
> This book (and its predecessor, *COMPUTE!'s Mastering the Apple IIGS Toolbox*) constantly reminds you to "follow the conventions" and "do it this way." If you don't follow the guidelines, you may find your work incompatible with future releases of the computer or operating system.
>
> You'll notice that few programmers obey all of the rules and suggestions mentioned in the guidelines. Just as with other aspects of life, some people don't pay attention. Apple Computer itself is one of the worst offenders and doesn't always pay attention to those warnings, either. Just ask anyone who owns a Mac II. Because Apple didn't follow its own rules, a good deal of its own software won't work on the Mac II.
>
> If you buy and read a copy of the Human Interface Guidelines, you'll notice that there are many recommendations that Apple never follows. The best advice is to do what they say and not what they do. Follow the guidelines and you will avoid trouble in the future.

What Are the Human Interface Guidelines?

Addison-Wesley has published a book written by Apple entitled *Human Interface Guidelines: The Apple Desktop Interface.* You can buy this book at your favorite bookstore (ISBN 0-201-17753-6). It's the latest rendition of an on-going project at Apple. While researching *Advanced Programming Techniques for Mastering the Apple IIGS Toolbox,* we located and mulled over one of the photocopied originals of the Human Interface Guidelines. Not much has changed since then. Only the list of contributing authors has grown longer. Still, most of the work can be attributed to Bruce Tognazzini (also lovingly called "Saint" Tognazzini). And before that, much of the philosophy on the interface came from work originally done at Xerox's Palo Alto Research Center (Xerox PARC).

Most of the beginning of the book is devoted to philosophizing and self-admiration of the Macintosh, mouse, and the desktop interface. Since you know how to point, click, and drag, that information is left out of this appendix.

Instead, you'll find the high points of the Human Interface Guidelines, all you really need to keep in step with what Apple likes to see in Apple programs. If you follow these guidelines, your program will be more compatible with other Apple IIGS and Macintosh programs. And Apple will like you. What more could you want?

The Desktop Environment

The desktop environment is the latest, supposedly best way for a computer to communicate with a human. It's called *visual communication.* Rather than typing names and commands, you do things visually with the mouse and with graphic icons which appear on the screen.

You might think that this setup would mean anyone could use an Apple computer immediately. You would be wrong. People still have hang-ups about computers. No matter how easy you make them, some people would have you throw pitchforks at them before they would use a computer.

The following are highlights of the guidelines:

- Every action on the desktop should be as simple and consistent as possible. The Human Interface Guidelines give the greatest weight to visual communication, simplicity, and clarity.

- Don't be rude to the user. Always provide a way out. When you are given the choice between doing something potentially dangerous and backing out, always make the default choice the way out. In other words, it should never be easy to do something stupid.
- Keep your desktop consistent. Changing screen modes is about the most unforgivable offense. True, the 320 mode is more colorful, and the 640 mode can display more text. Yet a word processor that uses one mode for one thing and the second mode for another would be dreadful. Users admire stability.
- Cut down on the dazzle. You can do amazing things with the Toolbox and QuickDraw, but try not to overwhelm the user with spectacular graphics and stereophonic sound. Look up the word *aesthetic* in the dictionary if you have trouble with what programmers call *creeping elegance.*

Programming for the Toolbox

In case you haven't noticed, all programs written for the Apple IIGS Toolbox follow a convention. They consist of a main event loop nested between setup and shutdown routines. (See Chapter 3 of *COMPUTE!'s Mastering the Apple IIGS Toolbox* for additional information.) This technique makes for better organization of your programs, making your programs easier to modify, as well (and incidentally, the code is easier to adapt for your other programs).

The following are a few concepts to keep in mind while designing and writing your programs:

- Implement what Apple calls *User Control* in your programs. Make the user choose what goes on. Don't make it appear that there is no way to control what the computer is doing.
- Provide the user with a complete list of options at any decision point. This is what separates desktop programs from IBM-type programs. In the IBM (command line interface) version of a program, it's up to you to remember what commands to type. With an Apple program, the user should see all the options available and then visually select one. Avoid hidden or secret options.
- When using an icon as a switch, make the icon closely resemble the action it invokes. For example, icons of an ImageWriter and LaserWriter can be used to choose a printer instead of an input box with the prompt *Enter printer:*.

- When you provide text (to explain a dialog box or amplify a choice, for instance), write a solid, meaningful description. Too many programmers opt to be overly cryptic with their text descriptions. But don't be overly simple with your text, either. Notice how *Send the contents of your document to the printer* is too simple, and *Dump File to Printer Device* is too complex, but *Print document: Chapter One?* is just right.

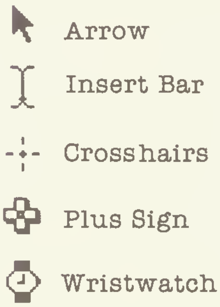
Mouse Traps

The guidelines go into great detail about use of the mouse, to the extent of discussing the algorithms used to select text with the mouse. Since this is an internal function of the Toolbox there's no need to repeat it here. Instead, the following are the mouse highlights of the guidelines:

- Using the mouse with your programs should be consistent with other desktop programs. Remember the standard mouse operations (pointing, clicking, dragging, double-clicking, and so on). Don't make up new mouse modes that could confuse the user.
- Though all Apple computers now have cursor-control keys on their keyboards, Apple demands that you never use the arrow keys as a replacement for the mouse. Never. You shouldn't even use the arrow keys to choose menu selections. (It should be pointed out, however, that Apple uses the cursor keys to imitate the mouse on the Macintosh.)

Though you can change the cursor's shape to just about anything (a pointing finger, for example), the following shapes are suggested for certain activities:

Figure A-1. Mouse Pointer (Cursor) Shapes



- The pointer is the most common default cursor.
- The I-beam is used for inserting and selecting text. The I-beam (or, if active, the pointer), disappears when the user starts typing.
- The crosshairs pointer is used to select graphic shapes for manipulation.
- The plus sign is used in some spreadsheet programs to select cells in the worksheet. It can also be used to select fields in an array. (The original Macintosh spreadsheet program, *Multiplan*, first employed the plus sign.)
- The tiny wristwatch stands for a pause as the machine does some work behind the scenes.

Pull-Down Menus

Menus are among the prime ingredients of the desktop. You should already know about menu titles and menu items and where they fit into the big picture. Keep in mind that the organization of menus and menu items (and command areas) is in your control.

Standard Menus

There are three menus most programs should have. For the sake of consistency, certain menu items should appear only in these menus. The standard menus are

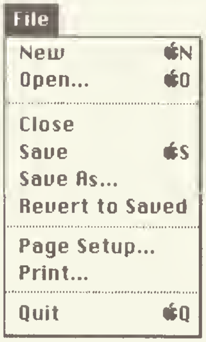
- The Apple menu
- The File menu
- The Edit menu

Text-based programs can also have Font, Style, and Size menus. However, Apple is less fussy about them.

- The Apple menu is always the first menu on the far left side of the menu bar. The first item at the top of this menu is an About. . . menu item used to display a dialog box telling about your program.
- Under the About. . . item come the various desk accessories installed in your SYSTEM/DESK.ACCS subdirectory. Also, you can put a Help item in the Apple menu and any configuration item or desk accessories specific to the application, such as a spelling checker for a word processor.
- The File menu contains all the items that deal with saving, loading, and creating data files. Aside from its allowances for opening, closing, and saving files, this menu also contains print options

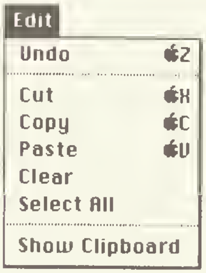
and the Quit option. Even if your program lacks any disk access, this is where the Quit option should go. The typical file menu appears as shown in Figure A-2.

Figure A-2. Graphic of File Menu


File
New N
Open... O
Close
Save S
Save As...
Revert to Saved
Page Setup...
Print...
Quit Q

- The final required menu is the Edit menu. A lot of emphasis is put on the cut-and-paste aspect of the desktop. Therefore, the Edit menu is considered important to all applications. (Even if your program doesn't need the items in the Edit menu, you might want them included for use by some desk accessories.)

Figure A-3. Graphic of Edit Menu


Edit
Undo Z
Cut X
Copy C
Paste V
Clear
Select All
Show Clipboard

One of the common items on the Edit menu is Select All. There is no key equivalent officially defined for the Select All item, although many applications seem to implement their own (usually Open Apple–A).

Other menus that might crop up from time to time, especially in text-oriented programs, are

- The Font menu
- The Size menu
- The Style menu

There are no hard-and-fast guidelines for these menus. If you have a crowded menu bar, you can combine two or three of them into one menu, or include all the options in a dialog box that looks like a Boeing 747 control panel.

Menu Items

The guidelines have the following suggestions for menu items:

- Menu items can be verbs used to describe an immediate reaction, or they can be adjectives used to describe some attribute of a selection:
  - With verb menu items, you choose a menu item and that task is carried out. If the program requires more input before the action can take place, the menu item should be followed by ellipses (. . .). If the item toggles a state on or off, a check mark should appear to indicate when the item is on, or you can choose to change the menu item's text—for example, *Inhale* could be changed to *Exhale.*
  - When menu items are adjectives (in font menus, for example), they should be descriptive words and adequately characterize what they change. Consider the opaqueness of a menu entry such as *Font* 2 when it is compared to something more descriptive, such as *Courier.*

A third type of menu, introduced with the Apple IIGS, is the color menu. This menu has no words, only the hues of colors available for changing selected items.

Commonly used menu items should be at the top of a menu, with less frequently used items at the bottom. Good examples are the Undo menu item commonly found at the top of the Edit menu, and the Quit menu item found at the bottom of the File menu.

Key Equivalents

You can assign key equivalents to just about any menu item. Be sure they make sense. Also consider that some actions are appropriate for the mouse and others are appropriate for the keyboard. A word processor is keyboard-intensive (although a mouse is great for editing). When users are typing, they'll find it more convenient to use a keyboard equivalent of a pull-down menu item than to grab the mouse, pull down the menu, and make the selection. On the other side of the coin, a paint program is a mouse-intensive piece of software. Having a keyboard-only command could be awkward.

Some of the older Macintosh communications programs lacked key equivalents entirely. This was because they used the Command key (later the Open Apple key) instead of the Control key to generate control codes. Apple IIGS communications programs have access to both the Open Apple–Command key and the Control key. So there is no reason to write a program devoid of Apple key equivalents.

The following keyboard equivalents must be used exclusively for the function described. Aside from these, you can assign whatever key equivalents your program might require:

Keyboard Equivalent	**Comment**	**Menu**
Open Apple–?	Help	Apple
Open Apple–C	Copy	Edit
Open Apple–N	New	File
Open Apple–O	Open	File
Open Apple–Q	Quit	File
Open Apple–S	Save	File
Open Apple–V	Paste	Edit
Open Apple–X	Cut	Edit
Open Apple–Z	Undo	Edit

Note that Open Apple–/ is considered the same as Open Apple–? (which is actually Open Apple–Shift–/). (See Chapter 8 for information on defining these keys.)

Less stringently obeyed are the following text-style key equivalents:

Keyboard Equivalent	**Comment**
Open Apple–B	Bold
Open Apple–I	Italic
Open Apple–P	Plain
Open Apple–U	Underline

A special-case Open Apple key equivalent is Open Apple–. (Open Apple–period). This key equivalent can be used to halt an action such as printing a document or a file listing in the *APW* shell. Apple implemented Open Apple–. because some Macintosh keyboards lacked an Esc key (normally Esc would be used). A few older programs may stick with the Open Apple–. convention. However, if you decide to implement an Esc cancel key in your programs, you might want to add Open Apple–. just to be well-received.

Dialog Boxes

Dialog boxes are actually special forms of windows. They are divided into modal and modeless types, as well as the special-case Alert boxes. The guidelines include the following information about dialog boxes:

- Dialog boxes should be placed in the center of the upper third of the screen. (The examples in this book were positioned in the center of the upper half because it was more aesthetically pleasing.)
- Alert boxes can be positioned so that their default button is in the same position as that occupied by the button that activated them. For example, this would allow the user to quickly cancel an operation without moving the mouse.
- A dialog box should always contain a message. It might describe what the dialog does or give some indication of what is happening. Don't crowd the text into the dialog. If you need more room, make the dialog box bigger.
- The most important and most commonly used items in a dialog box should be placed at the top, just as they are in the pull-down menu. Less frequently used items should be placed at the bottom. You can also place the more important items on the left side of the box, and the less important ones on the right.
- Remember to include in the dialog box a button that lets the user out.
- The OK button is associated with the Return key and the Cancel button is associated with the Esc key. Don't confuse the user by mixing these up.

There is such a creature as a dialog box without buttons. An example is a simple text box that displays a message and then disappears. One use for this sort of dialog is to inform the user how long an operation will take. For some reason, users don't mind waiting 15 minutes for an operation if the program is smart enough to tell them to do so.

Alerts

An alert dialog box is an example of a specific dialog with a specific use. In some cases, you may find that a simple beep of the speaker will replace an alert. For example, if a user clicks outside of a field, it's much faster to make the speaker *bonk* than to bring up a complete alert box.

Take advantage of the various alert stages. During your beta testing, you may discover that some alert boxes appear more often than others, indicating perhaps that a specific type of error is more likely to occur. If so, you may want to rethink your program's strategy. Ask questions of your beta testers to see if this happens.

The guidelines make the following suggestions:

- Keep alerts clean. Don't use radio buttons, long-scrolling text messages, check boxes, or other clutter. The typical alert box has an alert icon, a short message (or warning), and two buttons.
- The two buttons in an alert box typically allow the chosen action to continue or to be stopped. For example, an alert might display the message *Erase your hard disk?* The two buttons could be *Yes* and *No*, or even better, *Erase* and *Stop*. Typically, however, you should phrase your prompts so that it would be natural to supply buttons marked *OK* and *Cancel*.
- The default button in an alert box is always Cancel. The purpose of the alert is to warn of some impending danger. The default choice should always be to back away from the danger—in other words, make users really think about what they're doing.
- The alert message could be a system error, or something that your program can't handle. When this is the case, you may want to rethink your error-trapping routines and perhaps take the error-correcting decision out of the user's hands.

Notes on Sound and Color

The Apple IIGS comes with excellent sound and graphics. With the addition of the Mac II, sound and color have also been made available to the Macintosh line of computers.

The following are the guidelines on the use of sound and color in your programs. Generally speaking, the suggestions themselves are rather obvious, if you think about them. Listed below are only the high points.

Sound. The general thrust of the guidelines approach to sound is that sound should be used as an attention-getter. Use sound to say *Hey you!* should an application require immediate attention, or use it to alert the user that something is happening in the background. Other highlights:

- Try not to startle the user with sound.

- Different sounds can be used to herald entering and exiting certain modes. Of course, you may find these modal sounds annoying. It would be nice to include an option in your program for shutting off the noises (or for a volume control, at least).

Color. Color can be fun, and a great benefit to your programs. However, there are a few guidelines about the use of color. Most of these you can figure out on your own. For instance, an all-red foreground and background can make computing difficult. Still, some of the other guidelines are interesting and, when you pause to think about them, make sense.

- Different colors can be used in a number of ways. For example, you can color some text or a dialog box icon red to indicate something drastic. The color yellow can be used to show caution. Green is used to indicate *go* or *proceed.*
- Blue, especially light blue, is hard to see, and the guidelines recommend avoiding its use. However, an example of a good use of light blue would be providing rules or grids for a paint program; the blue is just faint enough to use as a reference.
- Use color to show how certain objects are grouped together, or to define separate areas.
- Keep the background light. A dark red background will make any foreground text difficult to see. Some programmers get carried away with color. Remember that users just want to use your program. Psychedelic colors went out with the sixties, along with love beads and sandalwood incense.

Above all, consider the application. Colored text looks good on the screen (and has probably sold more than one Apple IIGS). However, few people can print colored text. If the application is one that could use some color—such as a drawing, painting, or educational program—use it. But for text-intensive programs, think twice before splashing the screen with color, or at least provide the user with the option of choosing the colors to be used on the text display.

It should be noted that the terms *text* and *text display* have been tossed about freely in this appendix. True, the Apple IIGS does have a text display mode that can use different-colored backgrounds and letters. But the references to text are meant to include any textual material displayed on the graphics screen as well.

Summary

It goes without saying that a copy of the Apple Human Interface Guidelines will provide more detailed information than this appendix. However, the desktop environment is constantly changing. As Apple develops the IIGS and its other computers, and as programmers provide more interesting and intuitive applications, the guidelines will no doubt change. Just remember these two things:

- Users love to play with things.
- Above all, have fun with your programming.

Appendix B

Tool Sets in the Apple IIGS Toolbox

Table B-1. Tool Sets

Number	Tool Set Name	Version
$01	Tool Locator	$0201
$02	Memory Manager	$0200
$03	Miscellaneous	$0200
$04	QuickDraw II	$0202
$05	Desk Manager	$0202
$06	Event Manager	$0201
$07	Scheduler	$0200
$08	Sound Manager	$0201
$09	Apple DeskTop Bus	$0201
$0A	SANE	$0202
$0B	Integer Math	$0200
$0C	Text Tool Set	$0200
$0D	RAM Disk	$0200
$0E	Window Manager	$0201
$0F	Menu Manager	$0200
$10	Control Manager	$0202
$11	System Loader	—?—
$12	QuickDraw II Auxiliary	$0202
$13	Print Manager	$0102
$14	LineEdit	$0200
$15	Dialog Manager	$0200
$16	Scrap Manager	$0102
$17	Standard File	$0200
$18	Disk Utilities	—?—
$19	Note Synthesizer	$0100
$1A	Note Sequencer	—?—
$1B	Font Manager	$0201
$1C	List Manager	$0201

The high-order byte of the version number indicates the major release number and the low-order byte is the minor release. If bit 7 of the major release is set (bit 15 of the word), the release is a beta version. For example, $0201 (binary 0000 0010 0000 0001) indicates version 2.1, and $8101 (binary 1000 0001 0000 0001) indicates beta (prerelease) version 1.1.

The version numbers above apply to the ROM 01 release of the Apple IIGS as well as to the tool sets on System Disk version 3.1. Version numbers shown as —?— indicate tool sets which are not yet available.

TV.C Program

Since version numbers change as fast as the wind in Cupertino, Program B-1 (a C program) will generate a table just like the one printed above with the latest tool set version information for your system.

Program B-1. TV.C

```
/*-------------------------------------------*
 *                  TV.C                     *
 *     Displays all known toolset versions   *
 *-------------------------------------------*/


#include <types.h>
#include <prodos.h>
#include <intmath.h>
#include <locator.h>
#include <memory.h>
#include <misctool.h>
#include <texttool.h>
/*-----------------------------*
 *           Main              *
 *-----------------------------*/


Word UserID;             /* Our User ID */
Word Toolist[] = {
        12,              /* Tool count */
        14, 0,          /* Window Manager */
        15, 0,          /* Menu Manager */
        16, 0,          /* Control Manager */
        18, 0,          /* QD II Aux */
        19, 0,          /* Print Manager */
        20, 0,          /* Line Edit */
        21, 0,          /* Dialog Manager */
        22, 0,          /* Scrap Manager */
        23, 0,          /* Standard File */
        25, 0,          /* Note Synth */
        27, 0,          /* Font Manager */
        28, 0           /* List Manager */
};

QuitRec QParms = { NULL, 0 };   /* ProDOS 16 Quit parameter list */

main()
{
   TLStartUp   ();                    ErrChk();
   UserID      = MMStartUp ();        ErrChk();
   MTStartUp   ();                    ErrChk();
   WriteString ("\pLoading tools...");
   LoadTools   (Toolist);             ErrChk();
   WriteLine   ("\p");
   WriteLine   ("\p");

   ShowVers    ();                    /* Show Versions */

   MTShutDown  ();
   MMShutDown  (UserID);
   TLShutDown  ();
```

```
    QUIT        (&QParms);              /* Quit to ProDOS */

}


/*------------------------------*

 *    Handle Toolbox Errors     *

 *------------------------------*/


ErrChk() {  if (_toolErr) SysFailMgr(_toolErr, NULL);  }


/*------------------------------*

 *    Show Toolset Versions     *

 *------------------------------*/
  struct set {

      char    *name;

      word    id;

  } Toolset[] = {

      "\pTool Locator",             1,

      "\pMemory Manager",           2,

      "\pMiscellaneous Tools",      3,

      "\pQuickDraw II",             4,

      "\pDesk Manager",             5,

      "\pEvent Manager",            6,

      "\pScheduler",                7,

      "\pSound Manager",            8,

      "\pApple Desktop Bus",        9,

      "\pSANE",                     10,

      "\pInteger Math",             11,

      "\pText Toolset",             12,

      "\pRAM Disk",                 13,

      "\pWindow Manager",           14,

      "\pMenu Manager",             15,
    "\pControl Manager",        16,
    "\pSystem Loader",          17,
    "\pQuickDraw II Aux.",      18,
    "\pPrint Manager",          19,
    "\pLineEdit",               20,
    "\pDialog Manager",         21,
    "\pScrap Manager",          22,
    "\pStandard File",          23,
    "\pDisk Utilities",         24,
    "\pNote Synthesizer",       25,
    "\pNote Sequencer",         26,
    "\pFont Manager",           27,
    "\pList Manager",           28
};

#define ENTRIES (sizeof (Toolset) / sizeof (struct set))

char    *HexStr =   "\p$xxxx";
char    *Title  =   "\pNo.  Toolset Name        Version     ";
/*                   ¦''''¦''''¦''''¦''''¦''''¦''''¦''''¦''''¦ */

ShowVers()
{
    word    i;

    WriteString (Title);    WriteLine (Title);
    RepeatChar  ('=', 71);  WriteLine ("\p");

    for (i = 0; i < ENTRIES/2; ++i) {
        DoLine (i);
        RepeatChar (32, 6);
        DoLine (i + ENTRIES/2);
        WriteLine ("\p");
    }
}


DoLine(item)
word item;
{
    word    id, ver;

    id = Toolset[item].id;
    Int2Hex (id, HexStr+2, 2);
    HexStr[5] = HexStr[4] = 32;
    WriteString (HexStr);
    WriteString (Toolset[item].name);
    RepeatChar (32, 22 - (Toolset[item].name[0] & 0xff));
    asm {
        lda id
        ora #1024
        tax
        pha
        jsl dispatcher
        sta _toolErr
        pla
        sta ver
    }
    if (_toolErr)
        WriteString ("\p--?--");
    else {
        Int2Hex     (ver, HexStr+2, 4);

        WriteString (HexStr);

    }

}


RepeatChar (theChar, count)

char    theChar;

int     count;

{

    while (count--) WriteChar (theChar);

}
```

This program makes a handy utility to keep around on your *APW* system disk. To direct its output to a file or printer, use one of these *APW* shell commands:

```
tv >filename
tv >.printer
```

```
        Int2Hex     (ver, HexStr+2, 4);

        WriteString (HexStr);

    }

}


RepeatChar (theChar, count)

char    theChar;

int     count;

{

    while (count--) WriteChar (theChar);

}
```

This program makes a handy utility to keep around on your *APW* system disk. To direct its output to a file or printer, use one of these *APW* shell commands:

```
tv >filename
tv >.printer
```

Appendix C

Error Handling

There are several ways to deal with errors returned from the Toolbox. A blanket method has been shown in this book, one that's not really the best way to deal with potential errors. In fact, the method used by most Toolbox program examples in this book would be considered awful error trapping for a professional application.

Most of your programs should be smart enough to catch simple, common errors. Out-of-memory errors, disk I/O errors, and some Toolbox errors can easily be sidestepped. Your programs should make exceptions for the errors, recognize them, and deal with them in such a manner as to be transparent to the user. In other words, don't cop out on error handling.

ErrChk

Program C-1 is the error-checking code used in this book as the generic error handler, ErrChk. The problem with ErrChk is that it assumes every error returned from the Toolbox is a fatal, typically death-inducing error.

Program C-1. ErrChk in Machine Language

```
*------------------------------*

*    Handle Toolbox Errors     *

*------------------------------*


ErrChk  bcs Die                 ;Carry set if error


        rts                     ;Else, return


Die     pha                     ;Toolbox returns error in A

        pushlong #0             ;Use standard system death message

        _SysFailMgr             ;Get ready to slide apples back and forth
```

Program C-2 is the equivalent in C.

Program C-2. ErrChk in C

```
/*-----------------------------*
 *    Handle Toolbox Errors     *
 *-----------------------------*/

ErrChk()                             /* Check for error, die if so */
{
  if (_toolErr) SysFailMgr(_toolErr, nil);
}
```

Program C-3 is the equivalent in Pascal.

Program C-3. ErrChk in Pascal

```
{ *-----------------------------*
  *    Handle Toolbox Errors     *
  *-----------------------------* }

PROCEDURE ErrChk;                    { Check for error, die if so }
BEGIN
  IF IsToolError THEN
       SysFailMgr(ToolErrorNum, StringPtr(0));
END;
```

This error handler is called after every potential error-causing Toolbox function. All it checks is whether an error occurred. If so, the program bombs using the SysFailMgr call and tells you there's a fatal error. This is a very nondescript and somewhat crude method of error handling, albeit good for quick demonstrations and beta testing. But it doesn't take into consideration errors from which recovery is possible.

A Better Generic Error Handler

Documenting a procedure for each nonfatal Toolbox error would be complicated and would increase the size of this appendix to a full-blown chapter. Instead, the following example is provided to pique your curiosity.

This error-trapping routine (Program C-4) is designed to handle generic errors, and it can replace the ErrChk routine used throughout this book. Of course, it's a good idea to take care of nonfatal errors individually. This routine should be called only as a last-ditch effort. The code is listed in machine language. C and Pascal programmers can be inventive and code their own versions.

Program C-4. Fatal Error Handler

```
*-----------------------------*

*    Fatal Error Handler      *

*-----------------------------*


;only absolutely fatal errors are sent here


Die     pha                     ;Toolbox returns error in A, save it

        and #$FF00              ;get toolset number

        xba                     ;exchange MSB half of A-reg to LSB

        clc                     ;clear carry

        tay                     ;put a into Y

        lda #Table-28           ;offset from start of table

Die0    adc #28                 ;length of each entry

        dey                     ;dec count

        bne Die0                ;loop until the toolset is indexed


        phb                     ;push data bank twice

        phb                     ;(because phb pushes only a byte)

        pha                     ;push string's address

        _SysFailMgr             ;Get ready to slide apples back and forth
```

```
Table   str '       Tool Locator error $'

        str '     Memory Manager error $'

        str 'Miscellaneous Tools error $'

        str '       QuickDraw II error $'

        str '       Desk Manager error $'

        str '      Event Manager error $'

        str '          Scheduler error $'

        str '      Sound Manager error $'

        str '  Apple Desktop Bus error $'

        str '               SANE error $'

        str '       Integer Math error $'

        str '       Text Toolset error $'

        str '           RAM Disk error $'

        str '     Window Manager error $'

        str '       Menu Manager error $'
        str '    Control Manager error $'

        str '      System Loader error $'

        str '  QuickDraw II Aux. error $'

        str '      Print Manager error $'

        str '           LineEdit error $'

        str '     Dialog Manager error $'

        str '      Scrap Manager error $'

        str '      Standard File error $'

        str '     Disk Utilities error $'

        str '   Note Synthesizer error $'

        str '     Note Sequencer error $'

        str '       Font Manager error $'

        str '       List Manager error $'
```

This routine eliminates the *Fatal System Error* message and replaces it with something more specific. Rather than providing a two-byte hex number, this example translates the first number, representing the tool set, into a string. The actual error number is displayed after the dollar sign. So instead of

```
Fatal System Error —> $0E02
```

you are given

```
Window Manager error $0E02
```

Granted, this routine doesn't do anything the standard ErrChk routine didn't do, but it's more specific as to the type of error occurring. Again, a specific routine to deal with certain types of errors would be better.

This routine is still relatively simple. It would be easy to make it more elegant. For example, rather than padding each error string so that each takes up a fixed number of characters, you could use a table of pointers into variable length strings. It takes more source code to implement, but results in far less object code.

Appendix D

Error Codes

There are three types of errors you can receive from your computer. Unfortunately, it's sometimes hard to determine the origin of an error, though this appendix should help. The three types of errors you can receive are

- Fatal System errors
- ProDOS errors
- Toolbox errors

Fatal System errors are errors your programs won't be able to catch or wouldn't want to catch. Because these errors seem to pop up quite often for adventurous programmers such as yourself, they're listed here.

ProDOS errors are different from Toolbox errors in that their origins are in ProDOS and are not the result of any Toolbox mistakes you might have made. In fact, anyone who has programmed disk I/O or worked at all with any operating system is familiar with a DOS error. You can't build a decent program without DOS error trapping.

ProDOS errors are not incurable. For example, if your program returned the error *Disk Write Protected,* you could prompt the user to remove the write-protect tab or use another disk.

Toolbox errors aren't always fatal. In fact, quite a few are survivable (see Appendix C). However, more often than not, your program's error-handling routine may report a few of the more interesting ones. If your error-handling routine is smart, it can work around the error. Otherwise, make sure your program displays the error code so your users can report it back to you.

> Just to throw you a curve, there are some Toolbox function calls that result in errors originating from ProDOS. Yes, it's true. For example, the Tool Locator's LoadTools call or the Font Manager's FMStartUp function can return with an error flagged. An error code between $0001 and $00FF is a ProDOS 16 error. Error codes greater than $00FF are Toolbox errors.

Table D-1. Fatal System Errors

$01	Unclaimed interrupt
$0A	Volume control block unusable
$0B	File control block unusable
$0C	Block 0 allocated illegally
$0D	Interrupt occurred while I/O shadowing off
$11	Wrong OS version

Table D-2. Errors Returned from ProDOS

$00	No error
$01	Invalid call number
$07	ProDOS is busy
$10	Device not found
$11	Invalid device request
$25	Interrupt vector table full
$27	I/O error
$28	No device connected
$2B	Disk is write-protected
$2E	Disk switched, files open
$2F	Device not online
$30–$3F	Device-specific errors
$40	Invalid pathname
$42	File control block table full
$43	Invalid reference number
$44	Path not found
$45	Volume not found
$46	File not found
$47	Duplicate pathname
$48	Volume full
$49	Volume directory full
$4A	Version error
$4B	Unsupported storage type
$4C	EOF encountered, out of data
$4D	Position out of range
$4E	Access: file not rename-enabled
$50	File is open
$51	Directory structure damaged
$52	Unsupported volume type
$53	Invalid parameter
$54	Out of memory
$55	Volume control block full
$57	Duplicate volume
$58	Not a block device

$59	Invalid file level
$5A	Block number out of range
$5B	Illegal pathname change
$5C	Not an executable file
$5D	File system not available
$5E	Cannot deallocate /RAM
$5F	Return stack overflow
$60	Data unavailable

Table D-3. Errors Returned from the Toolbox

$0000	No error
$0001	Internal error, not enough arguments on the stack
$0002	Tool set wasn't activated (no StartUp call was made)
$0100	Unable to mount system startup volume
$0110	Bad tool set version number
$0201	Unable to allocate block
$0202	Illegal operation on an empty handle
$0203	Empty handle expected for this operation
$0204	Illegal operation on a locked or immovable block
$0205	Attempt to purge an unpurgeable block
$0206	Invalid handle given
$0207	Invalid User ID given
$0208	Operation illegal on block specified attributes
$0301	Bad input parameter
$0302	No device for input parameter
$0303	Task is already in the heartbeat queue
$0304	No signature in task header was detected
$0305	Damaged queue was detected during insert or delete
$0306	Task was not found during delete
$0307	Firmware task was unsuccessful
$0308	Detected damaged heartbeat queue
$0309	Attempted dispatch to a device that is disconnected
$030B	ID tag not available
$0401	QuickDraw already initialized
$0402	Cannot reset
$0403	QuickDraw is not initialized
$0410	Screen is reserved
$0411	Bad rectangle
$0420	Chunkiness is not equal
$0430	Region is already open
$0431	Region is not open
$0432	Region scan overflow

$0433	Region is full
$0440	Poly is already open
$0441	Poly is not open
$0442	Poly is too big
$0450	Bad table number
$0451	Bad color number
$0452	Bad scan line
$0510	Desk accessory is not available
$0511	Window pointer does not belong to the NDA
$0601	The Event Manager has already been started
$0602	Reset error
$0603	The Event Manager is not active
$0604	Bad event code number (greater than 15)
$0605	Bad button number value
$0606	Queue size greater than 3639
$0607	No memory for event queue
$0681	Fatal error: event queue is damaged
$0682	Fatal error: event queue handle is damaged
$0810	No DOC chip or RAM found
$0811	DOC address range error
$0812	No SAppInt call made
$0813	Invalid generator number
$0814	Synthesizer mode error
$0815	Generator busy error
$0817	Master IRQ not assigned
$0818	Sound Tools already started
$08FF	Fatal error: unclaimed sound interrupt
$0910	Command not completed
$0982	Busy, command pending
$0983	Device not present at address
$0984	List is full
$0B01	Bad input parameter
$0B02	Illegal character in input string
$0B03	Integer or long-integer overflow
$0B04	String overflow
$0E01	First word of parameter list is the wrong size
$0E02	Unable to allocate window record
$0E03	Bits 14–31 not clear in task mask
$1101	Segment or entry not found
$1102	Incompatible object module format (OMF) version
$1104	File is not a load file
$1105	System Loader is busy

$1107	File version error
$1108	UserID error
$1109	Segment number is out of sequence
$110A	Illegal load record found
$110B	Load segment is foreign

$1401	The LEStartUp call has already been made
$1402	Reset error
$1404	The desk scrap is too big

$150A	Bad item type
$150B	New item failed
$150C	Item not found
$150D	Not a modal dialog

$1610	Unknown scrap type

$1B01	Font Manager has already been started
$1B02	Can't reset Font Manager
$1B03	Font Manager is not active
$1B04	Family not found
$1B05	Font not found
$1B06	Font is not in memory
$1B07	System font cannot be purgeable
$1B08	Illegal family number
$1B09	Illegal size
$1B0A	Illegal name length
$1B0B	FixFontMenu never called

$1C01	Unable to create list control or scroll bar control

Appendix E

Event and TaskMaster Codes

Programs written for the Apple IIGS Toolbox center themselves on one event-oriented loop. Everything that happens in your programs is based upon a certain event—a mouse click, a drag, a selection. The Event Manager and its cousin the TaskMaster are at the heart of most DeskTop applications.

These events provide user input to your program. To determine which event has taken place (a mouse click, menu selection, or press of a key), your program makes a call to either the Event Manager's GetNextEvent function, or the Window Manager's TaskMaster function. Both of these procedures are covered within this book.

The Event Manager

The primary function of the Event Manager is GetNextEvent:

Function:	$0A06
Name:	GetNextEvent Returns the status of the event queue.
Push:	Result Space (W); Event Mask (W); Event Record (L)
Pull:	Logical Result (W)
Errors:	None
Comments:	If the Result is a logical true, an event is available. The event is then removed from the queue.

GetNextEvent deals with two items, the event mask and the event record. The event mask is used to scan only for specific types of events. The event record contains information about the event when GetNextEvent returns a logical true.

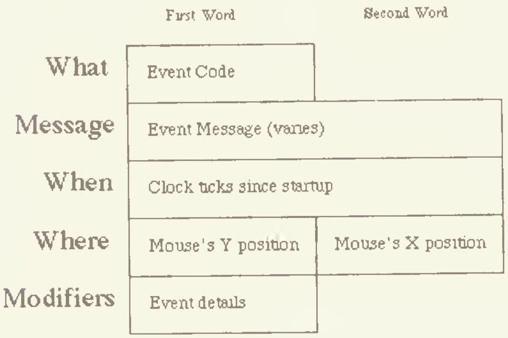
The event mask. The event mask is a word-sized value used to filter out certain types of events. By setting specific bits in the event mask, your program can direct GetNextEvent to return only the results of specific events. The following chart shows which bits in the event mask affect which events.

Table E-1. Bit in the Event Mask

Bit	Events Scanned for, if Set
0	Not used
1	Mouse-down events
2	Mouse-up events
3	Keyboard (key-down) events
4	Not used
5	Auto-key events
6	Update events
7	Not used
8	Activate events
9	Switch events
10	Desk accessory events
11	Device drive events
12	User-defined events
13	User-defined events
14	User-defined events
15	User-defined events

When GetNextEvent returns a true value, the event record will contain information detailing the event.

Figure E-1. The Event Record


First Word
Second Word
What
Event Code
Message
Event Message (varies)
When
Clock ticks since startup
Where
Mouse's Y position
Mouse's X position
Modifiers
Event details

The Event Record

The structure of the event record is as shown in Table E-2.

Table E-2. Structure of Event Record

Field	Size	Description
What	Word	Code describing event
Message	Long	Value or pointer providing more detail about the event
When	Long	Number of clock ticks since the computer was started
Where	Long	Two word values; the Y and X position of the mouse at the time of the event
Modifiers	Word	Describes the state of certain keys, the mouse button, and other information

What. The What field contains the event code. This describes which event took place. The events are numbered 0–15 (these are not bit values). The value found in the What field will be one of those shown in Table E-3.

Table E-3. Events in What Field

Event Code	Description
0	Null Event: Nothing has happened.
1	The mouse button was just pressed.
2	The mouse button has been released.
3	A key on the keyboard is being pressed.
4	Not used.
5	Auto-key event: a key is being held down.
6	Update event: a window is being changed, redrawn, sized, or its contents updated.
7	Not used.
8	Activate event: generated when a window becomes either active or inactive.
9	Switch event: activated when one program switches control to another.
10	Control–Open Apple–Esc has been pressed (this event is handled by the Desk Manager).
11	A device driver has generated an event.
12	User-defined (can be defined by your application).
13	User-defined.
14	User-defined.
15	User-defined.

Message. The Message field's value depends on the event code found in the What field.

Table E-4. Message Returned

Event Code	Message Field Contents
0	Undefined.
1	Button number (low-order word only).
2	Button number (low-order word only).
3	ASCII character (lowest byte only).
4	Undefined.
5	ASCII character (lowest byte only).
6	Window pointer.
7	Undefined.
8	Window pointer.
9	Undefined.
10	Undefined.
11	Value is returned from the device driver.
12	Value is returned from the user-defined application.
13	Value is returned from the user-defined application.
14	Value is returned from the user-defined application.
15	Value is returned from the user-defined application.

When. The When field contains the number of clock ticks since the computer was started. Each tick equals 1/60 second.

Where. The Where field gives the location of the mouse pointer at the time of the event, even if the event isn't mouse-oriented. The first word of the Where field contains the mouse's Y (vertical) position; the second word contains the mouse's X (horizontal) position.

Modifiers. The Modifiers field allows further description of the event pulled from the event queue.

Table E-5. Modifiers

Bit	Description
0	If set, the window pointed to in Message field is being deactivated; otherwise, the window is activated.
1	If set, the active window is changing from the system window to an application's window, or vice versa.
2	Not used.
3	Not used.
4	Not used.
5	Not used.
6	If set, mouse button number 1 is down.
7	If set, mouse button number 0 is down.
8	If set, the Open Apple key is down.
9	If set, a Shift key is down.

Bit	Description
10	If set, the Caps Lock key is down.
11	If set, the "option" (Solid Apple) key is down.
12	If set, the Control key is down.
13	If set, a key on the keypad is down.
14	Not used.
15	Not used.

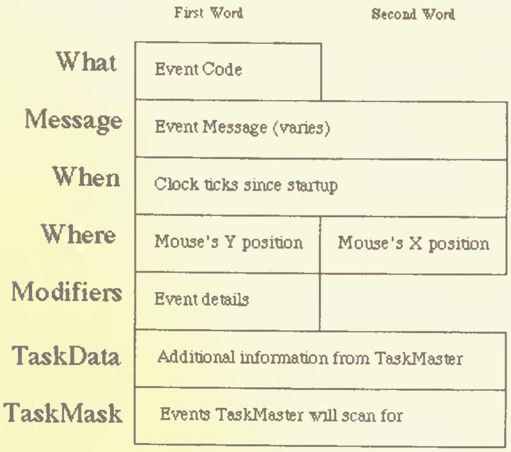
TaskMaster

TaskMaster, though a function of the Window Manager, is similar to GetNextEvent. It adds extra functions for managing windows and pull-down menus and, secretly, calls GetNextEvent internally:

Function: $1D0E
Name: TaskMaster
Returns status of the event queue as well as checks for certain window/menu events.
Push: Result Space (W); Event Mask (W); Event Record (L)
Pull: Extended Event Code (W)
Errors: $0E03

TaskMaster uses the same event mask as described above. It adds, however, two fields to the event record, TaskData and TaskMask:

Figure E-2. Event Record with TaskMaster Fields Added


First Word
Second Word
What
Event Code
Message
Event Message (varies)
When
Clock ticks since startup
Where
Mouse's Y position
Mouse's X position
Modifiers
Event details
TaskData
Additional information from TaskMaster
TaskMask
Events TaskMaster will scan for

The Event Record plus
TaskMaster Fields

Extended event codes. Unlike GetNextEvent, which returns a true or false value, TaskMaster returns either an event code or 0. When an event occurs, TaskMaster returns a value representing the event code. This code incorporates all the values found in the What field of the event record after a GetNextEvent function, plus 13 extended events.

Remember, the event codes are returned from the Toolbox when TaskMaster is called. You don't have to examine the What field of the Event Record, as is done with GetNextEvent, to determine which event took place.

The 13 extra values, or extended event codes, are shown in Table E-6.

Table E-6. Extended Event Codes

Event Code	Description
16	Mouse is in desk.
17	A menu item was selected.
18	Mouse is in the system window.
19	Mouse is in the content of a window.
20	Mouse is in drag.
21	Mouse is in grow.
22	Mouse is in goaway.
23	Mouse is in zoom.
24	Mouse is in info bar.
25	Mouse is in vertical scroll.
26	Mouse is in horizontal scroll.
27	Mouse is in frame.
28	Mouse is in drop.

TaskData. The two extra fields on the event record help to further describe the above codes. TaskData contains additional information about the extended event code. For the standard event codes 0–15, TaskData will be blank. But for the extended event codes 16–28, Task Data contains the values shown in Table E-7.

Table E-7. Meaning of TaskData

Code	TaskData Values
16	Not used
17	Not used
18	Not used
19	Not used
20	HOW = Menu ID, LOW = $0000

Code	TaskData Values
21	HOW = Menu ID, LOW = Menu Item
22	Window pointer
23	Window pointer
24	Window pointer
25	Window pointer
26	Window pointer
27	Window pointer
28	Window pointer

See examples from Chapters 8 and 9 on how this field is used.

TaskMask. The TaskMask field is similar to the event mask. It's used to filter out certain types of events monitored by the TaskMaster. These events are above and beyond those already filtered by the event mask. Both an event mask and a TaskMask are required by TaskMaster.

By setting specific bits in the TaskMask, your program can direct TaskMaster to return only the results of specific events. Table E-8 shows which bits in the TaskMask field affect which events. Note that bits 13–31 must always be set to 0, or an error results.

Table E-8. Bits in TaskMask

Bit	TaskMaster Scans for, if Set
0	MenuKey: menu item key equivalents
1	Update handling
2	FindWindow: mouse click in a window
3	MenuSelect: choosing a menu item
4	OpenNDA: new desk accessories in the Apple menu
5	System click
6	Drag window
7	Select window
8	Track goaway button
9	Track zoom button
10	Grow window
11	Allow scrolling
12	Handle special menu items
13–31	Must be set to 0

It's generally a good idea to set all the important bits. When this field is set to a value of $000003FFF, it will scan for and be able to handle all conceivable events.

Appendix F

QuickDraw II Color Information

In the current version of the Apple IIGS, the color tables used by QuickDraw II are stored at the following addresses. Each color table is $20 bytes long. (These address may change with future releases of the Apple IIGS ROMs):

Table F-1. Color Table Locations

Color Table	Address
0	$E19E00
1	$E19E20
2	$E19E40
3	$E19E60
4	$E19E80
5	$E19EA0
6	$E19EC0
7	$E19EE0
8	$E19F00
9	$E19F20
10	$E19F40
11	$E19F60
12	$E19F80
13	$E19FA0
14	$E19FC0
15	$E19FE0

Colors in the 320 mode. In the 320 mode, nibble positions for each color are as follows:

Table F-2. Color Nibble Positions

Color Value	Low Intensity	High Intensity
Blue	$0001	$000F
Green	$0010	$00F0
Red	$0100	$0F00

A color value of $0000 is black (all three colors are turned off). A color value of $0FFF is white (all three colors are at their highest intensity). Note how each color has 16 steps of intensity (from $0 to $F).

Table F-3. Standard Color Table in 320 Mode

Color Value	Color Number	Setting
Black	0	$0000
Dark Gray	1	$0777
Brown	2	$0841
Purple	3	$07C2
Blue	4	$000F
Dark Green	5	$0080
Orange	6	$0F70
Red	7	$0D00
Beige	8	$0FA9
Yellow	9	$0FF0
Green	10	$00E0
Light Blue	11	$04DF
Lilac	12	$0DAF
Periwinkle	13	$078F
Light Gray	14	$0CCC
White	15	$0FFF

Colors in the 640 mode. In the 640 mode, nibble positions for each color are as follows:

Table F-4. Color Nibble Positions

Color	Value
Blue	$000F
Green	$00F0
Red	$0F00

Unlike the 320 mode, there are only two values for each color in the 640 mode: $0 for off and $F for on.

Table F-5. Standard Color Table in 640 Mode

Color Value	Color Number	Setting
Black	0	$0000
Red	1	$0F00
Green	2	$00F0
White	3	$0FFF
Black	4	$0000
Blue	5	$000F
Yellow	6	$0FF0
White	7	$0FFF
Black	8	$0000
Red	9	$0F00

Color Value	Color Number	Setting
Green	10	$00F0
White	11	$0FFF
Black	12	$0000
Blue	13	$000F
Yellow	14	$0FF0
White	15	$0FFF

Index

abort interrupt 276
About . . . dialog box examples 229–35
alert box 191, 192–93, 210–17
Alert function 211
alert program example 214–17
alerts
  psychology of 214
  types of 210–11
alert stages 213
AlertTemplatePtr template 212
ALLOC_INTERRUPT ProDOS function 285
Apple computers, history of 11
Apple logo, menus and 117
Apple IIe emulation 19–20
Apple IIGS, how different from other Apples 10–11
APW (Apple Programmer's Workshop) C and ML kit 2
APW assembler 19, 68
  requirements 68
arguments, Toolbox and 52–53
assembler macros, ProDOS 16 and 332
BASIC programming language 2–3
bell, modifying 277–282
bit twiddling 14–15
blink rate, menu item, changing 137
block 28
book, how to use 4
books, other noteworthy 7
booting ProDOS 16 30–33, 331
BootInit function 51
Boot Loader 28–29
Boot ROM 28, 30
byte 5, 6
CASE ON *APW* directive 70
CautionAlert function 211
CDA (Classic Desk Accessories) 310, 311–16
  DOS and 311–312
CDA header 312
check box 249–51
  color table 250–51
  items 249
CloseDialog function 199, 204
CloseWindow function 151–52
closing
  dialog box 245
  Toolbox 56–57
  window 151–52
ClrHeartBeat function 301
color
  controls and 259–60
  programming standards for 382–83
  scroll bar and 256–57
colors
  menu bar 137–39
  QuickDraw II and 408–10
  radio button 253
  standard, 320 mode 139
command key equivalents, recommended 116
CompactMem function 107
COMPUTE!'s Mastering the Apple IIGS Toolbox 2
computer startup 26–27
Control Manager tool set 146, 242–44
  requirements for starting 243
controls 241–69
  dimming 264–65
  highlighting 265–66
  types of 244–45
conventions used in book 5–6
COPY ML directive 69
C programming language 2
CRC.ASM program 359–61
CRC.C program 350–59
CRC.PAS program 361–63
CtlShutDown 244
CtlStartUp call 243
cyclic redundancy checksum 349
DEALLOC_INTERRUPT ProDOS function 285
default button, importance of 194
DeleteMenu function 136–37
DeleteMItem function 136–37
DelHeartBeat function 300
desk accessories 309–28
DeskTop 77–94, 373–74
  menus and 114
DeskTop program, parts of 80–81
DeskTop programs, sample 81–94
dialog box 187–239
  closing 245
  controls 193–98
  modal 190, 192, 200–201

- modeless 190, 192, 217–25
- placing on screen 198–209
- planning for 191–93
- programming standards for 380–81
- types of 188
- very large 229

Dialog Manager 188–93
- requirements for starting 188–89

DialogSelect function 219
DialogShutdown function 189
DialogStartUp function 189
directory record 346–47
direct pages 53
disk
- contents, ProDOS 16 33–35
- error codes, list of 363–64
- tool sets 53–55

DisposeHandle function 107
dithering 12
DOC (Digital Oscillator Chip) 15–16
DrawMenuBar function 123
DrefCom 199
edit lines, scroll bar 257–58
equate files, *APW* assembler 65–66
ErrChk C program 392
ErrChk ML program 391
ErrChk Pascal program 392
error codes 396–400
- disk list of 363–64

error handling 391–95
- C 56
- Pascal 56

errors
- fatal system, list of 397
- ProDOS 16 and 334–35, 397–98
- Toolbox 55–56, 398–400

Event Manager 79, 401–5
event mask 401–2
event record 124, 403
false value 6
Fatal Error Handler ML program 393–95
file manipulation 344–50
filenames, ProDOS 160
Finder program 23, 36
FixMenuBar function 123
Font Manager tool set 330
function
- list 57–59
- number, tool set 44–45

functions
- ProDOS 16 335–36
- standard, tool set 50–51
- Toolbox 44
- tool set prefix 51

GetDItemValue function 197
GetNewDItem
- function 193, 205, 219, 245, 257
- template 205

GetNewID function 278, 293
GetNewModalDialog
- function 193, 257
- template 208

GetNextEvent function 401
GetVector function 283–84
Guidelines, Human Interface 115, 191, 371–83
handle 98
header files 67
HEARTBEAT.ASM program 301–5
HeartBeat task manager 299–301
HideDItem function 229
Human Interface Guidelines 115, 191, 371–83
icons
- alert 211
- defining 225–28

InsertMenu function 121–23, 135–36
InsertMItem function 135–36
interrupts 271–307
- abort 276
- action of 276–77
- Apple IIGS, types of 273–76
- enabling and disabling 287–88
- handler 272, 277–282, 287–96
- keyboard 298–99
- maskable 274–75
- nonmaskable 275
- ProDOS 16 and 284–86
- serial port and 273
- software 275
- sources, clearing 297–98
- vectors 282–84

IntSource function 287, 298
IsDialogEvent function 219
ItemColor dialog box parameter 198
ItemDescr dialog box parameter 196–98
ItemFlag dialog box parameter 198
item flags, setting 130–31
item handlers 126–27
Item ID, as index 126
ItemID dialog box parameter 193–94
ItemRect dialog box parameter 194–95
items
- check box 249
- Push Button 246
- radio button 252–53
- scroll bar 254–56

ItemType dialog box parameter 195–96
keyboard interrupts 298–99
key equivalents, programming standards for 378–79
Launcher program 36–37
launching applications 36–37, 40–42
load file types 41
LoadTools function 53–54
long word 5, 6

machine language. *See* ML
Macintosh emulation 10, 78
macros
  in source code 73–75
  list 70
  ML 69–75
MAGNIFY.NDA program 319–28
maskable interrupt 274–75
Mega II chip 19–20
memory
  additional 3, 22
  addressing 20–22
  block 95–96, 104–6
  handles, reusing 107
  management 95–111
  manager function list 108–11
  purge level 107
  relocating 98
  removing 106–7
  requesting 101–3
memory-block record 103–4
Memory Manager tool set 21, 31, 52, 95–111
  error codes 111
  starting 99
menu
  Apple logo and 117
  bar, drawing 123–24
  bar colors 137–39
  designing 116–20
  DeskTop and 114
  flags, setting 130
  ID 119–20
  installing 121–23
  item, changing 128–30
  item, changing blink rate 137
  item, renaming 132–34
  list 116–18
Menu Manager tool set 49, 114–43
  starting 121–22
menus, programming standards for 376–78
menus, pull-down 113–43
  renaming 134–35
  strings and 116, 120–21
  text style, changing 131–32
  title, unhighlighting 127–28
ML
  calling ProDOS 16 from 332
  calling Toolbox from 47–49
  modular programming 68–69
  support files and 68–75
  window naming and 159–61
ML programs, fatal-error handler 393–95
ModalDialog function 199, 204, 207, 208
MODEL.ASM program 81–87
MODEL.C program 87–90
MODEL.PAS program 90–93
modeless dialog box, example of use 218, 220–25
modem 273
modes and resolutions, chart of 12
modular programming, ML 68–69
MONDO.ASM program 168–73
  C language source for 173–77
  Pascal source for 178–82
mouse 79
  programming standards for 375–76
MtStartUp function 47–48
native mode, turning on 46–47
NDA (New Desk Accessories) 310, 316–19
  action codes 319
  DeskTop and 316
  header 317
  requirements for 316–17
NewDItem function 193, 202–3, 205, 219, 226, 245
NewHandle function 53, 101–2, 103–4, 278, 280–81
NewMenu function 116, 121
NewModalDialog function 201–2, 207
NewModelessDialog function 219
  parameters 219–20
NewWindow function 146–47, 149
nonmaskable interrupt 275
NoteAlert function 211
NUMCONV.CDA program 313–16
older chips, emulation of 17–19
opening window 149–51
operating system, ProDOS 16 22–23
panic button program examples 260–64
parameter list, ProDOS call 336–38
parameters, window record 153–56
parameter tables, ProDOS calls 338–44
permanent initialization files 32
planning for dialog boxes 191–93
port address, window 147
ProDOS 8 operating system 22–23, 330–31
  booting 28–30
ProDOS 16 operating system 22–23, 329–67
  assembler macros and 332
  booting 30–33, 331
  calling 331–32
  calling from C and Pascal 332–34
  calling from ML 332
  call parameter list 336–38
  calls, list of 338–44
  calls parameter tables 338–44
  disk contents 33–35
  errors and 334–35, 397–98
  filenames 160
  functions 335–36
  interrupts and 284–86
  Kernel Relocator 29
  Quit function 37–38
program, relocatable 40
program ID 39

programming hints, language-specific 61–75

programming standards
  for color 382–83
  for dialog boxes 380–81
  for key equivalents 378–79
  for menus 376–78
  for mouse 375–76
  for sound 381–82

programs
  how they work 25–42
  switching between 39–40

push buttons 245–49
  frame style 247

PushLong macro 71–73

QuickDraw II tool set 14, 49, 259

Quit call 39–40

Quit function, ProDOS 16 37–38

quit-parameter word 39–40

Quit Return Stack 39

radio button 251–53
  colors 253
  items 252–53

ramdisk 22

ReallocHandle function 107

"refertorial" approach of book 4

relocatable code 278

reply record 348

requirements for
  APW assembler 68
  NDA 316–17
  starting Control Manager 243
  using book 2

reset function 51

reset interrupt 275

screen, secret memory location of 10, 13

scroll bar 254–59
  edit lines 257–58

sector 28

serial port, interrupts and 273

SetBarColors function 138

SetDItemValue function 197

SetHeartBeat function 300

SetMenuFlag function 130

SetMenuTitle function 135

SetMItemBlink function 137

SetMItemFlag function 130–31

SetMItem function 133

SetMItemName function 133–34

SetVector function 282–84, 293, 298

SFAllCaps function 349

SFGetFile function 345–48

SFPutFile function 348–49

ShowDItem function 229

ShutDown function 51

65816 chip 17

software interrupt 275

sound 14–16
  programming standards for 381–82

spelling checker 218

Standard File Operations tool set 330, 344–50

standards, importance of 10

startup device 26

StartUp function, tool set 47, 51, 52–53

status function 51–52

StopAlert function 211

strings, menus and 116

style bit 131

super hi res 12, 14

support files 62–66
  C and 66–67
  ML and 68–75
  Pascal and 67–68
  tool set, list of 64–65

SYSTEM/DESK.ACCS subdirectory 310

system loader, Apple IIGS 31, 40

system menu bar 115

SYSTEM/TOOLS subdirectory 53

TaskMaster 80, 114, 124–26, 148–49, 152, 218, 405–7

template, dialog 198

temporary initialization files 32–33

terminal program 132

text style, menu, changing 131–32

TML Pascal 2, 67

Toolbox 3, 43–59
  calling 47–49
  calling from ML 47–49
  closing 56–57
  errors returned from 398–400
  functions, errors and 55–56
  importance of using 10
  starting 46–47

Tool Locator 330

tools, disk-based 44

tool set 44
  function list, menu 139–43
  interdependencies 49–50

tool sets
  disk 53–55
  list of 45, 384
  order in which to start 49–50

true value 6

TV.C program 385–90

type styles 119

UCSD Pascal 67

User ID 99–101

version, tool set 385

Version function 51

VGC (Video Graphics Controller) chip 11–12

voice, musical 15
wColor parameter, window record 162
wContDefProc 165–66
window 145–85
  closing 151–52
  color in 162–65
  contents 165–66
  controls and 244–45
  controls in 147–48
  naming 159–62
  opening 149–51
  record 146, 152–59, 162
Window Manager 80, 145–66
  starting 146
word 5, 6

Programming the Apple IIGS Toolbox

Advanced Programming Techniques for the Apple IIGS Toolbox provides you with the golden key to programming the Apple IIGS. Rare is the programming book that can claim to be both solidly packed with information and well written. This is one of those books.

Written by recognized IIGS authorities Morgan Davis and Dan Gookin, *Advanced Programming Techniques for the Apple IIGS Toolbox* delves into the intricacies of the powerful set of libraries known collectively as the Toolbox. You'll appreciate the equal treatment given to machine language, Pascal, and C and the organization that allows you to come to the book with a specific problem and leave with a realistic solution.

Here are just a few things you'll find in *Advanced Programming Techniques for the Apple IIGS Toolbox:*

- Many programming examples
- DeskTop programs
- Icons
- Interrupts
- ProDOS 16
- Human Interface Guidelines

Advanced Programming Techniques for the Apple IIGS Toolbox is the library of vital programming data that will earn its place next to your IIGS time and again.

$19.95

ISBN 0-87455-130-7
51995
9 780874 551303